# 前话 - 前端密码加密是否有意义?

首先我们明确知道走 HTTPS 是目前唯一负责的方式。但在 HTTP 环境下，无论如何都可能会被劫持流量，不管前端做不做加密都会被轻易成功登录。这个时候保护密码明文是否有意义?

首先，前端系统的控制权是完全在用户手里的，也就是说，前端做什么事情，用户有完全的控制权。如果是前端做过了 md5 加密，而后端没有做加密，一旦数据库泄露了，黑客就直接拿到了每个用户的密码 md5 值，由于黑客知道密码是在前端进行哈希的，所以他不需要爆破出该 md5 对应的原文是什么，而是直接修改客户端向服务器发出的请求，把密码字段换成数据库中 md5 就可以了，由于与数据库中记录一致，直接就会登录成功。这跟直接存储明文密码没有任何区别！所以不管前端是不是加密了密码，后台使用安全的哈希算法对内容再次转换是非常有必要的。（当然 md5 可能不再满足你的要求，需要用到 bcrypt）

其次，前端加密的原因是不让你的账户密码被泄露，因为劫持者可以拿着你的用户密码去碰撞其他资源，绝大多数人都是同一套账户密码。当然即使只能拿到加密后的密码，也可以采用 HASH 碰撞来逆推密码，不过这个成本就高了，而且不同的网站会有不同的加密算法，逆推不一定有意义。

所以一般第一步：前端使用 md5 来加密（除了 md5 还有 sha1.js，base64.js 等）：

```
npm install md5

//使用
var md5 = require('md5');

console.log(md5('message'));
```

第二步、后端收到密码后，也要先进行加密，一般为了添加安全性，会额外在密文的基础上加盐再做一次加密然后存入数据库。

# 加密

加密是以某种算法改变原有的信息数据，使得未授权用户即使获得了已加密信息，因不知解密的方法，无法得知信息真正的含义，通过这种方式提高网络数据传输的安全性，加密算法常见的有哈希算法、HMAC 算法、签名、对称性加密算法和非对称性加密算法，加密算法也分为可逆和不可逆，比如 md5 就是不可逆加密，只能暴力破解（撞库），我们在 NodeJS 开发中就是直接使用这些加密算法，crypto 模块提供了加密功能，包含对 OpenSSL 的哈希、HMAC、加密、解密、签名以及验证功能的一整套封装，核心模块，使用时不需安装。

## 哈希算法

哈希算法也叫散列算法，用来把任意长度的输入变换成固定长度的输出，常见的有 md5、sha1 等。

### md5

md5 是开发中经常使用的算法之一，官方称为摘要算法，具有以下几个特点：

- 不可逆；
- 不管加密的内容多长，最后输出的结果长度都是相等的；
- 内容不同输出的结果完全不同，内容相同输出的结果完全相同。

当连续被 md5 加密 3 次以上时就很难被破解了，所以使用 md5 一般会进行多次加密。

```
const crytpo = require("crytpo");

let md5 = crytpo.createHash("md5"); // 创建 md5
let md5Sum = md5.update("hello"); // 加密
let result = md5Sum.digest(); // 获取加密后结果
// let result = md5Sum.digest("hex");
// let result = md5Sum.digest("Base64");

console.log(result); // <Buffer 5d 41 40 2a bc 4b 2a 76 b9 71 9d 91 10 17 c5 92>
```

update 方法的返回值就是 this，即当前实例，所以支持链式调用。

digest 方法参数用于指定加密后的返回值的格式，不传参默认返回加密后的 Buffer，常用的参数有 hex 和 Base64，hex 代表十六进制，加密后长度为 32，Base64 的结果长度为 24，以 == 结尾。

> 注意：一个 crypto 实例只能调用 digest 一次，所有每次调用 digest 都需要创建个新 crypto 实例：

```
const crypto = require('crypto');

router.post("/signup", function(req, res) {
  const md5Pass = crypto.createHash('md5').update(xxx).digest('hex');
});

router.post("/signin", function(req, res) {
  const md5Pass = crypto.createHash('md5').update(xxx).digest('hex');
});
```

# Hmac 算法

Hmac 算法又称加盐算法，是将哈希算法与一个密钥结合在一起，用来阻止对签名完整性的破坏，同样具备 md5加密的几个特点。

```
const crytpo = require("crytpo");

// crytpo.createHmac 参数一为加密的算法，常用 sha1 和 sha256，第二个参数为密钥
let hmac = crytpo.createHmac("sha1", "panda");
let result = hmac.update("hello").digest("Base64");

console.log(result); // 7spMLxN8WJdcEtQ8Hm/LR9pUE3YsIGag9Dcai7lwioo=
```

安全性高于 md5，通过密钥来加密，不知道密钥无法破解，缺点是密钥传输的过程容易被劫持，可以通过一些生成随机密钥的方式避免。

## 创建随机密钥的方法

安装 openSSH 客户端，并通过命令行生成存储密钥的文件:

```
openssl genrsa -out rsa_private.key 1024
```

openssl genrsa 代表生成密钥，-out 代表输出文件，rsa_private.key 代表文件名，1024 代表输出密钥的大小。

读取密钥文件配合加盐算法加密:

```
const fs = require("fs");
const crytpo = require("crytpo");
const path = require("path");

let key = fs.readFileSync(path.join(__dirname, "/rsa_private.key"));
let hmac = crytpo.createHmac("sha256", key);

let result = hmac.update("hello").digest("Base64");

console.log(result); // bmi2N+6kwgwt5b+U+zSgjL/NFs+GsUnZmcieqLKBy4M=
```

# 对称性加密

对称性加密是发送数据时使用密钥和加密算法进行加密，接收数据时需要使用相同的密钥和加密算法的逆算法（解密算法）进行解密，也就是说对称性加密的过程是可逆的，crytpo 中使用的算法为 blowfish。

```
const fs = require("fs");
const crypto = require("crypto");
const path = require("path");

let key = fs.readFileSync(path.join(__dirname, "/rsa_private.key"));

// 加密
let cipher = crypto.createCipher("blowfish", key);
cipher.update("hello");

// final 方法不能链式调用
let result = cipher.final("hex");
console.log(result); // 3eb9943113c7aa1e

// 解密
let decipher = crypto.createDecipher("blowfish", key);
decipher.update(result, "hex");

let data = decipher.final("utf8");
console.log(data); // hello
```

加密使用 crypto.createCipher 方法，解密使用 crypto.createDecipher 方法，但是使用的算法和密钥必须相同，需要注意的是解密过程中 update 中需要在第二个参数中指定加密时的格式，如 hex，在 final 还原数据时需要指定加密字符的编码格式，如 utf8。

注意：使用对称性加密的字符串有长度限制，不得超过 7 个字符，否则虽然可以加密成功，但是无法解密。

缺点：密钥在传输过程中容易被截获，存在安全风险。

# 非对称性加密

非对称性加密相也是可逆的，较于对称性加密要更安全，消息传输方和接收方都会在本地创建一对密钥，公钥和私钥，互相将自己的公钥发送给对方，每次消息传递时使用对方的公钥加密，对方接收消息后使用他的的私钥解密，这样在公钥传递的过程中被截获也无法解密，因为公钥加密的消息只有配对的私钥可以解密。

使用 openSSH 对之前生成的私钥 rsa_private.key 产生一个对应的公钥:

```
openssl rsa -in rsa_private.key -pubout -out rsa_public.key
```

```
const fs = require("fs");
const crypto = require("crypto");
const path = require("path");

// 获取公钥和私钥
let publicKey = fs.readFileSync(path.join(__dirname, "/rsa_public.key"));
let privateKey = fs.readFileSync(path.join(__dirname, "/rsa_private.key"));

// 加密
let secret = crytpo.publicEncrypt(publicKey, Buffer.from("hello"));

// 解密
let result = crytpo.provateDecrypt(privateKey, secret);

console.log(result); // hello
```

# 签名

签名与非对称性加密非常类似，同样有公钥和私钥，不同的是使用私钥加密，对方使用公钥进行解密验证，以确保这段数据是私钥的拥有者所发出的原始数据，且在网络中的传输过程中未被修改。

```
const fs = require("fs");
const crypto = require("crypto");
const path = require("path");

// 获取公钥和私钥
let publicKey = fs.readFileSync(path.join(__dirname, "rsa_public.key"),
"ascii");
let privateKey = fs.readFileSync(path.join(__dirname, "rsa_private.key"),
"ascii");

// 生成签名
let sign = crypto.createSign("RSA-SHA256");
sign.update("panda");
let signed = sign.sign(privateKey, "hex");

// 验证签名
let verify = crypto.createVerify("RSA-SHA256");
verify.update("panda");
let verifyResult = verify.verify(publicKey, signed, "hex");

console.log(verifyResult); // true
```

生成签名的 sign 方法有两个参数，第一个参数为私钥，第二个参数为生成签名的格式，最后返回的 signed 为生成的签名（字符串）。

验证签名的 verify 方法有三个参数，第一个参数为公钥，第二个参数为被验证的签名，第三个参数为生成签名时的格式，返回为布尔值，即是否通过验证。

使用场景：经常用于对 cookie 签名返回浏览器，当浏览器访问同域服务器将 cookie 带过来时再进行验证，防止 cookie 被篡改和 CSRF 跨站请求伪造。

# 案例：密码加密的处理方式

## 一、扩展(一般加密过程)

1、利用随机数随机生成多少位数

2、利用可逆加密把第一步的生成的随机数加密 可逆加密有Base64和Hex加密(具体自己百度)

3、将第二步加密好的随机数与我们真实密码拼接在一起

4、将第三步进行加密(MD5)

5、将第四步进行可逆加密

6、将第二步与第五步生成的拼接成密码

```
// 随机数
function random(randomFlag, min, max) {
  var str = "",
    range = min,
    arr = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd',
'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
'u', 'v', 'w', 'x', 'y', 'z', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J',
'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'];

  // 随机产生
  if(randomFlag) {
    range = Math.round(Math.random() * (max-min)) + min;
  }
  for(var i=0; i < range; i++) {
    var pos = Math.round(Math.random() * (arr.length-1));
    str += arr[pos];
  }
  return str;
}
module.exports = random;
```

```
const express = require("express");
const router = express.Router();
const crypto = require("crypto");
// 引入数据库文件
const db = require("./../module/db");
// 引入生成随机数的
const random = require("./../utils/random");
// 引入 baset64 文件，或者 npm install --save js-base64
const Base64 = require("./../utils/base64");

// 注册
router.post("/regist", (req,res)=> {
  let { username, password } = req.body;

  // 1.生成8位的随机数
  let randomword = random(false,8);

  // 2.对生成的随机数 base64 编码
```

```
  let base64 = new Base64();
  let base64Random = base64.encode(randomWord);

  // 3.将第 2 步的值与密码拼接
  let newPas = base64Random + password;

  // 4.将第 3 步的进行 md5 加密
  let md5 = crypto.createHash("md5");
  let md5Pas = md5.update(newPas).digest("hex");

  // 5.将第 4 步进行 base64 编码
  let base64Md5 = base64.encode(md5Pas);

  // 6.将第 2 步与第 5 步拼接
  let lastPassword = base64Random + base64Md5;

  db("insert into users(username,password) values(?,?)", [ username,
lastPassword ], (err,data)=> {
    if (err) {
      res.send("注册失败");
    }
    console.log(data);
    if (data) {
      res.send("注册成功");
    }
  });
});

// 登录
router.post("/login", (req,res)=> {
  let { username, password } = req.body;

  db("select * from users where username = ?",[ name ], (err, data)=> {
    if (err) {
      return res.send("发生错误");
    }
    if (data) {
      // 与注册的加密顺序相反
      // 1.从数据库获取到的密码截取前面随机数通过 base64 编码后的结果
      let base64Random = data[0].password.substring(0,12);

      // 2.将第 1 步的结果与用户输入的密码拼接
      let newPas = base64Random + password;

      // 3.将第 2 步的结果进行 md5 加密
      let md5 = crypto.createHash("md5");
      let md5Pas = md5.update(newPas).digest("hex");

      // 4.将第 3 步进行 base64 编码
      let base64 = new Base64();
      let base64Md5 = base64.encode(md5Pas);

      // 5.将第 1 步与第 4 步拼接
      let lastPassword = base64Random + base64Md5;
      if (data[0].password === lastPassword) {
        res.send("登录成功");
      } else {
        res.send("用户名或密码错误");
```

```
      }
    }
  });
});

module.exports = router;
```