**如何存储密码才是安全的？**

密码存储有几种方式：

* 直接存储密码明文m
* 存储密码明文的哈希值hash(m)
* 存储密码明文的加盐哈希 hash(m+salt)，这里的salt可以是用户名，手机号等，但必须保证每个用户的salt都不一样才是安全的。

如果数据库被入侵。  
第一方式，明文存储，无安全性可言。  
第二种方式，虽然是入侵者得到的是hash值，但由于**彩虹表**的存在，也很容易批量还原出密码明文来。  
只有第三种方式才是相对安全的。

### 彩虹表不是 密码-->明文 的简单存储

要从c=hash(m)逆向得到原始明文m，有三种办法：

* 暴力破解法：时间成本太高。
* 字典法：提前构建一个“明文->密文”对应关系的一个大型数据库，破解时通过密文直接反查明文。但存储一个这样的数据库，空间成本是惊人的。
* 构建彩虹表：在字典法的基础上改进，以时间换空间。是现在破解哈希常用的办法。

最早只知道用 MD5 来存用户的账号的密码，但其实这非常不安全，2004年，证实MD5算法无法防止碰撞（collision），因此不适用于安全性认证，如[SSL](https://baike.baidu.com/item/SSL/320778" \t "_blank)公开密钥认证或是[数字签名](https://baike.baidu.com/item/%E6%95%B0%E5%AD%97%E7%AD%BE%E5%90%8D/212550" \t "_blank)等用途。

而所用到的哈希函数，深入挖掘，也发现并不简单……

# 一、普通的 Hash 函数[#](https://www.cnblogs.com/xjnotxj/p/12716981.html" \l "3352441405)

哈希（散列）函数是什么就不赘述了。

### 1、不推荐[#](https://www.cnblogs.com/xjnotxj/p/12716981.html" \l "4186805912)

RC4, MD4, MD5, SHA-0, SHA-1, DES, 2DES 等

### 2、推荐[#](https://www.cnblogs.com/xjnotxj/p/12716981.html" \l "4108318445)

SHA-2(SHA-256, SHA-384, SHA-512)、SHA-3、Blake2 等

美国国家标准和技术协会（NIST）宣布，2010 年后开始逐步取消 SHA-1 作为安全哈希算法的资格，取而代之的是其更强大的变异算法：SHA-224、SHA-256、SHA-384 和 SHA-512。无论是否遵循 NIST 的标准，至少使用 SHA-256 算法加密密码总是好的。

# 二、应对普通哈希容易被破解的策略[#](https://www.cnblogs.com/xjnotxj/p/12716981.html" \l "2676734755)

就像攻与矛的互相增强，哈希函数哪怕用到 SHA-3 以上，都还是有被轻易破解的风险。于是我们有其他额外的办法来解决这个问题。

### 1、加盐（salt）[#](https://www.cnblogs.com/xjnotxj/p/12716981.html#2859235834)

加盐就是对目标字段哈希前，拼接上另一个字段（salt）。

注：盐值加到字段之前较为普遍。

加盐对防彩虹表很有效。

注意点：

* 盐不能太短
* 盐不能重复使用（否则一破解，所有的都遭殃）
* 盐随机变化（例如，虽用户名不重复，但用户名不能拿来当盐）

盐的本质是将无差别攻击转化为针对性攻击。

### 1.1、【拓展】针对 salt 的另一种做法 —— HMAC[#](https://www.cnblogs.com/xjnotxj/p/12716981.html#2872993792)

HMAC（Keyed-Hashing for Message Authentication）其实也是一种特殊的加盐，只是这个 salt 用更安全的**密钥**代替了。

### 2、慢哈希[#](https://www.cnblogs.com/xjnotxj/p/12716981.html" \l "2593988199)

高端的显卡（GPU）和定制的硬件可以每秒进行数十亿次哈希计算，因此这类攻击依然可以很高效。为了降低攻击者的效率，我们可以使用慢哈希，即迭代进行**很多次哈希运算**。

那么迭代**多少次**比较安全呢？来自 NIST 官方的建议：

* 2000 年 9 月，建议迭代一千次
* 2015 年 - 2018年，建议迭代一万次
* 2017 年 6 月，建议迭代**十万次**

# 三、密码哈希函数（Password Hash）[#](https://www.cnblogs.com/xjnotxj/p/12716981.html#3744687759)

密码哈希函数（Password Hash）可以用来应对普通哈希容易被破解的问题（也用到了上面所提到的两个策略）。

**下面列举的顺序是按照时间顺序，安全程度和推荐指数也逐级递增。**

### 1、PBKDF2[#](https://www.cnblogs.com/xjnotxj/p/12716981.html#3634309327)

比较老，很少有人用了，略。

### 2、Bcrypt[#](https://www.cnblogs.com/xjnotxj/p/12716981.html" \l "3229112731)

这是我司目前用的。（不过有过时的隐患，建议换掉）

##### （1）介绍[#](https://www.cnblogs.com/xjnotxj/p/12716981.html" \l "3208236975)

bcrypt 是由 Niels Provos 和 DavidMazières 基于 Blowfish 密码设计的密码哈希函数，于 1999 年在 USENIX 上提出。

bcrypt 函数是 OpenBSD 和其他系统（包括某些 Linux 发行版，例如 SUSE Linux）的默认密码哈希算法。

##### （2）使用（Node.js）[#](https://www.cnblogs.com/xjnotxj/p/12716981.html#4117358359)

安装：npm i bcryptjs

bcryptjs 跟 C++ 的 bcrypt 兼容，但因为是**纯 JavaScript 编写**的，因此速度较慢（约 30％）。

用法：

Copy

Sync 方法（Async 方法略）：

const bcryptjs = require('bcryptjs');

// 1、生成 安全因子

const salt = bcrypt.genSaltSync(10);

// 2、执行 哈希函数

const password = bcryptjs.hashSync(plainPassword, bcryptjs.genSaltSync(salt));

// 另一种方法：快速执行

const SALT\_FACTOR = 10;

const password = bcryptjs.hashSync(plainPassword, bcryptjs.genSaltSync(SALT\_FACTOR));

// 3、比较是否相等

bcryptjs.compareSync(plainPassword, password);

注：代码里出现的安全因子，值的大小决定了哈希函数会有多慢。（即慢哈希）

### 3、Scrypt[#](https://www.cnblogs.com/xjnotxj/p/12716981.html" \l "3131824352)

没用过，略。

### 4、Argon2[#](https://www.cnblogs.com/xjnotxj/p/12716981.html#2496249135)

##### （1）介绍[#](https://www.cnblogs.com/xjnotxj/p/12716981.html" \l "1142777812)

2013 年 NIST（美国国家标准与技术研究院）邀请了一些密码学家一起，举办了密码哈希竞赛 PHC（Password Hashing Competition）。Argon2 在 2015 年 7 月赢得了冠军。

大赛列出了参赛算法可能面临的攻击手段：

* 哈希算法破解（原值还原、哈希碰撞等）；
* 查询表/彩虹表攻击；
* CPU 优化攻击；
* GPU、FPGA、ASIC 等专用硬件攻击；
* 旁路攻击；

##### （2）使用（Node.js）[#](https://www.cnblogs.com/xjnotxj/p/12716981.html#1381646463)

1、准备

Copy

You can skip this section if the prebuilt binaries work for you.

You MUST have a node-gyp global install before proceeding with install, along with GCC >= 5 / Clang >= 3.3. On Windows, you must compile under Visual Studio 2015 or newer.

node-argon2 works only and is tested against Node >=10.0.0.

---

OSX

To install GCC >= 5 on OSX, use homebrew:

$ brew install gcc

Once you've got GCC installed and ready to run, you then need to install node-gyp, you must do this globally:

$ npm install -g node-gyp

Finally, once node-gyp is installed and ready to go, you can install this library, specifying the GCC or Clang binary to use:

$ CXX=g++-6 npm install argon2

NOTE: If your GCC or Clang binary is named something different than g++-6, you'll need to specify that in the command.

2、安装

npm i argon2

3、使用

Copy

const argon2 = require('argon2');

(async () => {

try {

// const hash = await argon2.hash("password");

// 更多选项（以下都是默认值）

const hash = await argon2.hash("password", {

type: argon2.argon2i,

hashLength: 32, // 哈希函数输出的字节长度(请注意，生成的哈希是使用Base64编码的，因此长度将增加约1/3)

timeCost : 3, // 时间成本是哈希函数使用的通过次数（迭代次数）

memoryCost: 2 \*\* 16, // 默认 4096（单位 KB，即 4MB）

parallelism :1, //用于计算哈希值的线程数量。每个线程都有一个具有memoryCost大小的内存池

})

console.log("hash", hash)

const is = await argon2.verify(hash, "password")

console.log("is", is) // true

} catch (err) {

console.error("err", err)

}

})()

##### （3）参数[#](https://www.cnblogs.com/xjnotxj/p/12716981.html" \l "3967712396)

1、type：

* argon2d 更快且对GPU攻击具有高度抵抗力，这对于加密货币很有用
* argon2i 速度较慢且可以抵御权衡攻击，因此首选用于密码哈希和密钥派生
* argon2id 是上述内容的混合组合，可以抵抗GPU和权衡攻击

因为我们是用于密码的 hash，用默认的 argon2i 即可。

2、（慢）哈希相关参数

① memoryCost 内存开销，它定义了内存的使用情况

好的起点是 0.75 \*（RAM / number\_of\_users) 起步。

② parallelism 并行程度，它定义了线程的数量

最佳起点是内核数。

③ timeCost 时间开销，它定义了执行的时间

建议在系统上运行它，并确定与内存和处理器使用时间限制相匹配的最大参数。

如前所述，本质是**在安全性和可用性之间取得平衡**。

3、其他参数

salt：默认值是未设置，将生成加密安全的随机盐。  
saltLength：默认16。

version：您不应更改此设置，因为最新版本更强大。

### 5、密码哈希是如何解决普通哈希容易被破解的问题[#](https://www.cnblogs.com/xjnotxj/p/12716981.html" \l "2870640992)

上面介绍了 二、应对普通哈希容易被破解的策略 ，我们可以看看密码哈希是如何运用并符合这些策略的。

##### （1）针对 salt[#](https://www.cnblogs.com/xjnotxj/p/12716981.html#4016277649)

密码哈希使用 CSPRNG（Cryptographically Secure Pseudo-Random Number Generator）密码学安全伪随机数生成器生成盐。

CSPRNG 是加密安全（Cryptographically Secure）的，（加密安全的意思即）意味着用它产生的随机数**更加随机**，且不可预测。

普通的计算机随机数算法并不是很随机。

注：**盐值本身就在存在于哈希后的字符串中（其实还可能包括版本、慢哈希迭代次数等），当调用跟明文比对的方法时，模块内部会提取出盐值进行验证。**

##### （2）针对 慢哈希[#](https://www.cnblogs.com/xjnotxj/p/12716981.html" \l "1174001934)

Bcryoy 的**安全因子**和 Argon2的 **timeCost** 参数，都是针对慢哈希的配置。

### 6、结论[#](https://www.cnblogs.com/xjnotxj/p/12716981.html" \l "2054841443)

我司使用的 Bcrypt 其实在今年（2020年），已经不安全了，**推荐至少使用 Scrypt，有条件上 Argon2**。

# 四、常见问题[#](https://www.cnblogs.com/xjnotxj/p/12716981.html" \l "350436361)

问1：我用我自己实现哈希算法，不用公开现成的，越古怪越好，坏人不就猜不到了吗？

答：不建议。

首先介绍下密码学上的柯克霍夫原则（Kerckhoffs's principle，也称为柯克霍夫假说、公理、或定律），由奥古斯特·柯克霍夫在 19 世纪提出：**即使密码系统的任何细节已为人悉知，只要密匙（key，又称密钥或秘钥）未泄漏，它也应是安全的。**信息论的发明者克劳德·香农则改成说：“敌人了解系统”，这样的说法则称为香农箴言。

基于这个原则：

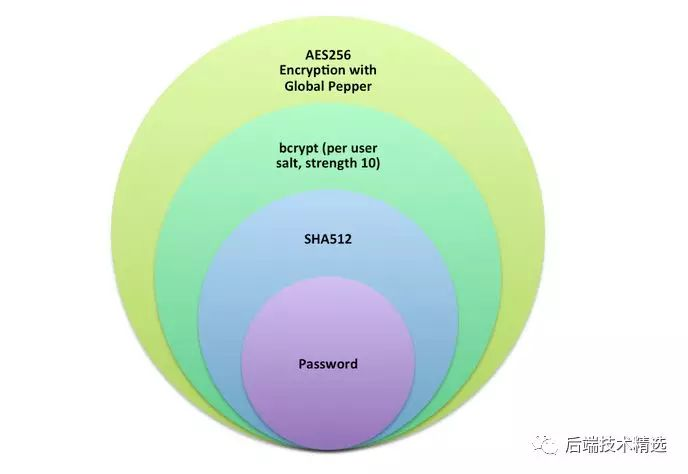
* 1、你自己实现的再古怪，毕竟你不是密码专家，很难确保不被坏人破解（可能自己实现后看似复杂，实际更容易破解了）。
* 2、如果自己包含哈希算法的代码泄露，它很脆弱，难保不会被坏人破解。

问2：既然现在都是 https，前端传给后端的明文密码，就懒得加哈希了，可以吗？

还是建议前端也进行哈希（虽然前端的哈希算法容易暴露）。不要漏掉任何一个环节。

**五、实操[#](https://www.cnblogs.com/xjnotxj/p/12716981.html?utm_source=tuicool&utm_medium=referral" \l "1969327135)**

Dropbox 公司曾公开分享过自己对用户账号的密码加密的策略，使用了三层加密：

[](https://img2020.cnblogs.com/blog/896608/202004/896608-20200416141959818-873499930.png)

**1、password**[**#**](https://www.cnblogs.com/xjnotxj/p/12716981.html?utm_source=tuicool&utm_medium=referral#4158958735)

即明文密码。

**2、SHA512**[**#**](https://www.cnblogs.com/xjnotxj/p/12716981.html?utm_source=tuicool&utm_medium=referral#2932366314)

在 bcrypt 前做 SHA512，是因为有些 bcrypt 实现会把散列值长度截至 72 字节，从而降低了密码的熵值，而有的则允许变长密码，这样容易受到 DoS 攻击。使用 SHA512 散列可以得到固定长度的 512 字节散列值，避免了上述的两个问题。

*”而有的则允许变长密码，这样容易受到 DoS 攻击“*，这句话我不是很理解，待写。

**3、bcrypt[#](https://www.cnblogs.com/xjnotxj/p/12716981.html?utm_source=tuicool&utm_medium=referral" \l "743220547)**

上面说过，不赘述了。

**4、AES256**[**#**](https://www.cnblogs.com/xjnotxj/p/12716981.html?utm_source=tuicool&utm_medium=referral#1846255104)

AES256 会用到密钥，俗称胡椒粉（pepper）。密钥需要被单独存储，最好存储在外部系统：如物理上隔离的服务端、甚至特殊的硬件设备（如 YubiHSM） 。

这里的 AES256 也可以用 HMAC 代替。不过前者安全性更好些。

Ref

<https://www.cnblogs.com/xjnotxj/p/12716981.html>

具体介绍可以看我之前一篇：[《破解另一家网站的反爬机制 & HMAC 算法》](https://www.cnblogs.com/xjnotxj/p/11934756.html)