作者：黎连文@哈希数字

V1.00 Date:2018/07/30

奇技淫巧

|  |  |
| --- | --- |
| 类别 | 内容 |
| 关键词 | 奇技淫巧 |
| 摘 要 | 奇技淫巧 |

**修订历史**

|  |  |  |
| --- | --- | --- |
| 版本 | 日期 | 原因 |
| V1.00 | 2018/11/14 | 创建文档 |
|  |  |  |

目 录

[1. 加盐密码哈希：如何正确使用 1](#_Toc530438607)

[1.1 为什么密码需要进行哈希？ 2](#_Toc530438608)

[1.2 如何破解哈希加密 3](#_Toc530438609)

[1.3 加盐 6](#_Toc530438610)

[1.4 常见问题 17](#_Toc530438611)

[1.5 PHP PBKDF2 密码哈希代码 24](#_Toc530438612)

[1.6 Java PBKDF2 密码哈希代码 28](#_Toc530438613)

[1.7 ASP.NET(C#) PBKDF2 密码哈希代码 35](#_Toc530438614)

[1.8 Ruby(on Rails) PBKDF2 密码哈希代码 38](#_Toc530438615)

# 加盐密码哈希：如何正确使用

如果你是Web开发者，你很可能需要开发一个用户账户系统。这个系统最重要的方面，就是怎样保护用户的密码。存放帐号的数据库经常成为入侵的目标，所以你必须做点什么来保护密码，以防网站被攻破时发生危险。最好的办法就是对密码进行**加盐哈希**，这篇文章将介绍它是如何做到这点。

在对密码进行哈希加密的问题上，人们有许多争论和误解，这大概是由于网络上广泛的误传吧。密码哈希是一件非常简单的事情，但是依然有很多人理解错误了。本文阐述的并不是进行密码哈希唯一正确的方法，但是会告诉你为什么这样是正确的。

**郑重警告：**如果你在试图编写自己的密码哈希代码，**赶紧停下来！**那太容易搞砸了。即使你受过密码学的高等教育，也应该听从这个警告。这是对所有人说的：**不要自己写加密函数！**安全存储密码的难题现在已经被解决了，请使用[phpass](http://www.openwall.com/phpass/)或者本文给出的一些源代码。

如果因为某些原因你忽视了上面那个红色警告，请翻回去好好读一遍，我是认真的。这篇文章的目的不是教你研究出自己的安全算法，而是讲解为什么密码应该被这样储存。

下面一些链接可以用来快速跳转到本文的各章节。

1. [为什么密码需要进行哈希？](http://blog.jobbole.com/61872/#toc1)
2. [如何破解哈希加密](http://blog.jobbole.com/61872/#toc2)
3. [加盐](http://blog.jobbole.com/61872/#toc3)
4. [无效的哈希方法](http://blog.jobbole.com/61872/#toc4)
5. [恰当使用哈希加密](http://blog.jobbole.com/61872/#toc5)
6. [常见问题](http://blog.jobbole.com/61872/#toc6)

这里也给出了一些基于BSD许可的哈希函数源代码：

* [PHP Source Code](http://blog.jobbole.com/61872/#php)
* [Java Source Code](http://blog.jobbole.com/61872/#java)
* [ASP.NET (C#) Source Code](http://blog.jobbole.com/61872/#csharp)

* [Ruby (on Rails) Source Code](http://blog.jobbole.com/61872/" \l "ruby" \o "Ruby (on Rails) Source Code)

## 为什么密码需要进行哈希？

hash("hello") = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824  
hash("hbllo") = 58756879c05c68dfac9866712fad6a93f8146f337a69afe7dd238f3364946366  
hash("waltz") = c0e81794384491161f1777c232bc6bd9ec38f616560b120fda8e90f383853542

哈希算法是一个单向函数。它可以将任何大小的数据转化为定长的“指纹”，并且无法被反向计算。另外，即使数据源只改动了一丁点，哈希的结果也会完全不同（参考上面的例子）。这样的特性使得它非常适合用于保存密码，因为我们需要加密后的密码无法被解密，同时也能保证正确校验每个用户的密码。

在基于哈希加密的账户系统中，通常用户注册和认证的流程是这样的：

1. 用户注册一个帐号
2. 密码经过哈希加密储存在数据库中。只要密码被写入磁盘，任何时候都不允许是明文
3. 当用户登录的时候，从数据库取出已经加密的密码，和经过哈希的用户输入进行对比
4. 如果哈希值相同，用户获得登入授权，否则，会被告知输入了无效的登录信息
5. 每当有用户尝试登录，以上两步都会重复

在第4步中，永远不要告诉用户到底是用户名错了，还是密码错了。只需要给出一个大概的提示，比如“无效的用户名或密码”。这可以防止攻击者在不知道密码的情况下，枚举出有效的用户名。

需要提到的是，用于保护密码的哈希函数和你在数据结构中学到的哈希函数是不同的。比如用于实现哈希表这之类数据结构的哈希函数，它们的目标是快速查找，而不是高安全性。只有**加密哈希函数**才能用于保护密码，例如SHA256，SHA512，RipeMD和WHIRLPOOL。

也许你很容易就认为只需要简单地执行一遍加密哈希函数，密码就能安全，那么你大错特错了。有太多的办法可以快速地把密码从简单哈希值中恢复出来，但也有很多比较容易实现的技术能使攻击者的效率大大降低。黑客的进步也在激励着这些技术的进步，比如这样一个网站：你可以提交一系列待破解的哈希值，并且在不到1秒的时间内得到了结果。显然，简单哈希加密并不能满足我们对安全性的需求。

那么下一节会讲到几种常用的破解简单哈希加密的办法。

## 如何破解哈希加密

**字典攻击和暴力攻击**

Dictionary Attack  
Trying apple : failed  
Trying blueberry : failed  
Trying justinbeiber : failed  
...  
Trying letmein : failed  
Trying s3cr3t : success!

Brute Force Attack  
Trying aaaa : failed  
Trying aaab : failed  
Trying aaac : failed  
...  
Trying acdb : failed  
Trying acdc : success!

• 破解哈希加密最简单的办法，就是去猜，将每个猜测值哈希之后的结果和目标值比对，如果相同则破解成功。两种最常见的猜密码的办法是**字典攻击**和**暴力攻击**。

• 字典攻击需要使用一个字典文件，它包含单词、短语、常用密码以及其他可能用作密码的字符串。其中每个词都是进过哈希后储存的，用它们和密码哈希比对，如果相同，这个词就是密码。字典文件的构成是从大段文本中分解出的单词，甚至还包括一些数据库中真实的密码。然后还可以对字典文件进行更进一步的处理使它更有效，比如把单词中的字母替换为它们的“形近字”（hello变为h3110）。

• 暴力攻击会尝试每一个在给定长度下各种字符的组合。这种攻击会消耗大量的计算，也通常是破解哈希加密中效率最低的办法，但是它最终会找到正确的密码。因此密码需要足够长，以至于遍历所有可能的字符串组合将耗费太长时间，从而不值得去破解它。

• 我们没有办法阻止字典攻击和暴击攻击，尽管可以降低它们的效率，但那也不是完全阻止。如果你的密码哈希系统足够安全，唯一的破解办法就是进行字典攻击或者暴力遍历每一个哈希值。

**查表法**

Searching: 5f4dcc3b5aa765d61d8327deb882cf99: FOUND: password5  
Searching: 6cbe615c106f422d23669b610b564800: not in database  
Searching: 630bf032efe4507f2c57b280995925a9: FOUND: letMEin12  
Searching: 386f43fab5d096a7a66d67c8f213e5ec: FOUND: mcd0nalds  
Searching: d5ec75d5fe70d428685510fae36492d9: FOUND: p@ssw0rd!

查表法对于破解一系列算法相同的哈希值有着无与伦比的效率。主要的思想就是**预计算**密码字典中的每个密码，然后把哈希值和对应的密码储存到一个用于快速查询的数据结构中。一个良好的查表实现可以每秒进行数百次哈希查询，即使表中储存了几十亿个哈希值。

如果你想更好地体验查表法的速度，尝试使用CrackStation的[free hash cracker](https://crackstation.net/)来破解下图中四个SHA256加密的哈希值吧。

c11083b4b0a7743af748c85d343dfee9fbb8b2576c05f3a7f0d632b0926aadfc  
08eac03b80adc33dc7d8fbe44b7c7b05d3a2c511166bdb43fcb710b03ba919e7  
e4ba5cbd251c98e6cd1c23f126a3b81d8d8328abc95387229850952b3ef9f904  
5206b8b8a996cf5320cb12ca91c7b790fba9f030408efe83ebb83548dc3007bd

**反向查表法**

Searching for hash(apple) in users' hash list... : Matches [alice3, 0bob0, charles8]  
Searching for hash(blueberry) in users' hash list... : Matches [usr10101, timmy, john91]  
Searching for hash(letmein) in users' hash list... : Matches [wilson10, dragonslayerX, joe1984]  
Searching for hash(s3cr3t) in users' hash list... : Matches [bruce19, knuth1337, john87]  
Searching for hash(z@29hjja) in users' hash list... : No users used this password

这种方法可以使攻击者同时对多个哈希值发起字典攻击或暴力攻击，而不需要预先计算出一个查询表。

首先攻击者构造一个基于密码-用户名的一对多的表，当然数据需要从某个已经被入侵的数据库获得，然后猜测一系列哈希值并且从表中查找拥有此密码的用户。通常许多用户可能有着相同的密码，因此这种攻击方式也显得尤为有效。

**彩虹表**

彩虹表是一种在时间和空间的消耗上找寻平衡的破解技术。它和查表法很类似，但是为了使查询表占用的空间更小而牺牲了破解速度。因为它更小，于是我们可以在一定的空间内存储更多的哈希值，从而使攻击更加有效。能够破解任何8位及以下长度MD5值的彩虹表已经[出现](http://www.freerainbowtables.com/en/tables2/)了。

下面我们会讲到一种让查表法和彩虹表都失去作用的技术，叫做加盐。

## 加盐

hash("hello") = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824  
hash("hello" + "QxLUF1bgIAdeQX") = 9e209040c863f84a31e719795b2577523954739fe5ed3b58a75cff2127075ed1  
hash("hello" + "bv5PehSMfV11Cd") = d1d3ec2e6f20fd420d50e2642992841d8338a314b8ea157c9e18477aaef226ab  
hash("hello" + "YYLmfY6IehjZMQ") = a49670c3c18b9e079b9cfaf51634f563dc8ae3070db2c4a8544305df1b60f007

查表法和彩虹表只有在所有密码都以相同方式进行哈希加密时才有效。如果两个用户密码相同，那么他们密码的哈希值也是相同的。我们可以通过“随机化”哈希来阻止这类攻击，于是当相同的密码被哈希两次之后，得到的值就不相同了。

比如可以在密码中混入一段“随机”的字符串再进行哈希加密，这个被字符串被称作盐值。如同上面例子所展示的，这使得同一个密码每次都被加密为完全不同的字符串。为了校验密码是否正确，我们需要储存盐值。通常和密码哈希值一起存放在账户数据库中，或者直接存为哈希字符串的一部分。

盐值并不需要保密，由于随机化了哈希值，查表法、反向查表法和彩虹表都不再有效。攻击者无法确知盐值，于是就不能预先计算出一个查询表或者彩虹表。这样每个用户的密码都混入不同的盐值后再进行哈希，因此反向查表法也变得难以实施。

下面讲讲我们在实现加盐哈希的过程中通常会犯哪些错误。

**错误一：短盐值和盐值重复**

最常见的错误就是在多次哈希加密中使用相同的盐值或者太短的盐值。

**盐值重复**

每次哈希加密都使用相同的盐值是很容易犯的一个错误，这个盐值要么被硬编码到程序里，要么只在第一次使用时随机获得。这样加盐的方式是做无用功，因为两个相同的密码依然会得到相同的哈希值。攻击者仍然可以使用反向查表法对每个值进行字典攻击，只需要把盐值应用到每个猜测的密码上再进行哈希即可。如果盐值被硬编码到某个流行的软件里，可以专门为这个软件制作查询表和彩虹表，那么破解它生成的哈希值就变得很简单了。

用户创建账户或每次修改密码时，都应该重新生成新的盐值进行加密。

**短盐值**

如果盐值太短，攻击者可以构造一个查询表包含所有可能的盐值。以只有3个ASCII字符的盐值为例，一共有95x95x95=857,375种可能。这看起来很多，但是如果对于每个盐值查询表只包含1MB最常见的密码，那么总共只需要837GB的储存空间。一个不到100美元的1000GB硬盘就能解决问题。  
同样地，用户名也不应该被用作盐值。尽管在一个网站中用户名是唯一的，但是它们是可预测的，并且经常重复用于其他服务中。攻击者可以针对常见用户名构建查询表，然后对用户名盐值哈希发起进攻。

为了使攻击者无法构造包含所有可能盐值的查询表，盐值必须足够长。一个好的做法是使用和哈希函数输出的字符串等长的盐值，比如SHA256算法的输出是256bits(32 bytes)，那么盐值也至少应该是32个随机字节。

**错误二：两次哈希和组合哈希函数**

（译注：此节标题原文中的Wacky Hash Functions直译是古怪的哈希函数，大概是由于作者不认可这种组合多种哈希函数的做法，为了便于理解，本文还是翻译为组合哈希函数）

这节讲述了另一种对密码哈希的误解：使用组合哈希函数。人们经常不由自主地认为将不同的哈希函数组合起来，结果会更加安全。实际上这样做几乎没有好处，仅仅造成了函数之间互相影响的问题，甚至有时候会变得更加不安全。永远不要尝试发明自己的加密方法，只需只用已经被设计好的标准算法。有的人会说使用多种哈希函数会使计算更慢，从而破解也更慢，但是还有其他的办法能更好地减缓破解速度，后面会提到的。

这里有些低端的组合哈希函数，我在网上某些论坛看到它们被推荐使用：

* md5(sha1(password))
* md5(md5(salt) + md5(password))
* sha1(sha1(password))
* sha1(str\_rot13(password + salt))
* md5(sha1(md5(md5(password) + sha1(password)) + md5(password)))

不要使用其中任何一种。

注意：这节内容是有争议的。我已经收到的大量的邮件，为组合哈希函数而辩护。他们的理由是如果攻击者不知道系统使用的哪种哈希函数，那么也就很难预先为这种组合构造出彩虹表，于是破解起来会花费更多的时间。

诚然，攻击者在不知道加密算法的时候是无法发动攻击的，但是不要忘了[Kerckhoffs’s principle](https://en.wikipedia.org/wiki/Kerckhoffs%27s_principle)，攻击者通常很容易就能拿到源码（尤其是那些免费或开源的软件）。通过系统中取出的一些密码-哈希值对应关系，很容易反向推导出加密算法。破解组合哈希函数确实需要更多时间，但也只是受了一点可以确知的因素影响。更好的办法是使用一个很难被并行计算出结果的迭代算法，然后增加适当的盐值防止彩虹表攻击。

当然你实在想用“标准的”组合哈希函数，比如HMAC，也是可以的。但如果只是为了使破解起来更慢，那么先读读下面讲到的密钥扩展。

创造新的哈希函数可能带来安全问题，构造哈希函数的组合又可能带来函数间互相影响的问题，它们带来的一丁点好处和这些比起来真是微不足道。显然最好的做法是使用标准的、经过完整测试的算法。

**哈希碰撞**

哈希函数将任意大小的数据转化为定长的字符串，因此其中一定有些输入经过哈希计算之后得到了相同的结果。加密哈希函数的设计就是为了使这样的碰撞尽可能难以被发现。随着时间流逝，密码学家发现攻击者越来越容易找到碰撞了，最近的例子就是MD5算法的碰撞已经确定被发现了。

碰撞攻击的出现表明很可能有一个和用户密码不同的字符串却和它有着相同的哈希值。然而，即使在MD5这样脆弱的哈希函数中找到碰撞也需要耗费大量的计算，因此这样的碰撞“意外地”在实际中出现的可能性是很低的。于是站在实用性的角度上可以这么说，加盐MD5和加盐SHA256的安全性是一样的。不过可能的话，使用本身更安全的哈希函数总是好的，比如SHA256、SHA512、RipeMD或者WHIRLPOOL。

**正确的做法：恰当使用哈希加密**

本节会准确讲述应该如何对密码进行哈希加密。其中第一部分介绍最基本的要素，也是在哈希加密中一定要做到的；后面讲解怎样在这个基础上进行扩展，使得加密更难被破解。

**基本要素：加盐哈希**

**忠告：你不仅仅要用眼睛看文章，更要自己动手去实现后面讲到的“让密码更难破解：慢哈希函数”。**

在前文中我们已经看到，利用查表法和彩虹表，普通哈希加密是多么容易被恶意攻击者破解，也知道了可以通过随机加盐的办法也解决这个问题。那么到底应该使用怎样的盐值呢，又如何把它混入密码？

盐值应该使用**基于加密的伪随机数生成器（Cryptographically Secure Pseudo-Random Number Generator – CSPRNG）**来生成。CSPRNG和普通的随机数生成器有很大不同，如C语言中的rand()函数。物如其名，CSPRNG专门被设计成用于加密，它能提供高度随机和无法预测的随机数。我们显然不希望自己的盐值被猜测到，所以一定要使用CSPRNG。下面的表格列出了当前主流编程语言中的CSPRNG方法：

|  |  |
| --- | --- |
| **Platform** | **CSPRNG** |
| PHP | [mcrypt\_create\_iv](http://php.net/manual/en/function.mcrypt-create-iv.php), [openssl\_random\_pseudo\_bytes](http://php.net/manual/en/function.openssl-random-pseudo-bytes.php) |
| Java | [java.security.SecureRandom](http://docs.oracle.com/javase/6/docs/api/java/security/SecureRandom.html) |
| Dot NET (C#, VB) | [System.Security.Cryptography.RNGCryptoServiceProvider](http://msdn.microsoft.com/en-us/library/system.security.cryptography.rngcryptoserviceprovider.aspx) |
| Ruby | [SecureRandom](http://rubydoc.info/stdlib/securerandom/1.9.3/SecureRandom) |
| Python | [os.urandom](http://docs.python.org/library/os.html) |
| Perl | [Math::Random::Secure](http://search.cpan.org/~mkanat/Math-Random-Secure-0.06/lib/Math/Random/Secure.pm) |
| C/C++ (Windows API) | [CryptGenRandom](http://en.wikipedia.org/wiki/CryptGenRandom) |
| Any language on GNU/Linux or Unix | Read from [/dev/random or /dev/urandom](http://en.wikipedia.org/wiki/dev/random) |

对于每个用户的每个密码，盐值都应该是独一无二的。每当有新用户注册或者修改密码，都应该使用新的盐值进行加密。并且这个盐值也应该足够长，使得有足够多的盐值以供加密。一个好的标准的是：盐值至少和哈希函数的输出一样长；盐值应该被储存和密码哈希一起储存在账户数据表中。

**存储密码的步骤**

1. 使用CSPRNG生成一个长度足够的盐值
2. 将盐值混入密码，并使用**标准的**加密哈希函数进行加密，如SHA256
3. 把哈希值和盐值一起存入数据库中对应此用户的那条记录

**校验密码的步骤**

1. 从数据库取出用户的密码哈希值和对应盐值
2. 将盐值混入用户输入的密码，并且使用同样的哈希函数进行加密
3. 比较上一步的结果和数据库储存的哈希值是否相同，如果相同那么密码正确，反之密码错误

文章最后有几个加盐密码哈希的代码实现，分别使用了PHP、C#、Java和Ruby。

**在Web程序中，永远在服务器端进行哈希加密**

如果你正在开发一个Web程序，你可能会疑惑到底在哪进行加密。是使用JavaScript在用户的浏览器上操作呢，还是将密码“裸体”传送到服务器再进行加密？

即使浏览器端用JavaScript加密了，你仍然需要在服务端再次进行加密。试想有个网站在浏览器将密码经过哈希后传送到服务器，那么在认证用户的时候，网站收到哈希值和数据库中的值进行比对就可以了。这看起来比只在服务器端加密安全得多，因为至始至终没有将用户的密码明文传输，但实际上不是这样。

问题在于，从客户端来看，经过哈希的密码逻辑上成为用户真正的密码。为了通过服务器认证，用户只需要发送密码的哈希值即可。如果有坏小子获取了这个哈希值，他甚至可以在不知道用户密码的情况通过认证。更进一步，如果他用某种手段入侵了网站的数据库，那么不需要去猜解任何人的密码，就可以随意使用每个人的帐号登录。

这并不是说你不应该在浏览器端进行加密，但是如果你这么做了，一定要在服务端再次加密。在浏览器中进行哈希加密是个好想法，不过实现的时候注意下面几点：

• 客户端密码哈希并**不能代替**HTTPS（SSL/TLS）。如果浏览器和服务器之间的连接是不安全的，那么中间人攻击可以修改JavaScript代码，删除加密函数，从而获取用户密码。

• 有些浏览器不支持JavaScript，也有的用户禁用了浏览器的JavaScript功能。为了最好的兼容性，你的程序应该检测JavaScript是否可用，如果答案为否，需要在服务端模拟客户端的加密。

• 客户端哈希同样需要加盐，很显然的办法就是向服务器请求用户的盐值，但是不要这么做。因为这给了坏蛋一个机会，能够在不知道密码的情况下检测用户名是否有效。既然你已经在服务端对密码进行了加盐哈希，那么在客户端把用户名（或邮箱）加上网站特有的字符串（如域名）作为盐值是可行的。

**让密码更难破解：慢哈希函数**

加盐使攻击者无法采用特定的查询表和彩虹表快速破解大量哈希值，但是却不能阻止他们使用字典攻击或暴力攻击。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，我们可以使用一种叫做**密钥扩展**的技术。

这种技术的思想就是把哈希函数变得很慢，于是即使有着超高性能的GPU或定制硬件，字典攻击和暴力攻击也会慢得让攻击者无法接受。最终的目标是把哈希函数的速度降到足以让攻击者望而却步，但造成的延迟又不至于引起用户的注意。

密钥扩展的实现是依靠一种CPU密集型哈希函数。不要尝试自己发明简单的迭代哈希加密，如果迭代不够多，是可以被高效的硬件快速并行计算出来的，就和普通哈希一样。应该使用标准的算法，比如[PBKDF2](http://en.wikipedia.org/wiki/PBKDF2)或者[bcrypt](http://en.wikipedia.org/wiki/Bcrypt)。[这里](https://defuse.ca/php-pbkdf2.htm)可以找到PBKDF2在PHP上的一种实现。

这类算法使用一个安全因子或迭代次数作为参数，这个值决定了哈希函数会有多慢。对于桌面软件或者手机软件，获取参数最好的办法就是执行一个简短的性能基准测试，找到使哈希函数大约耗费0.5秒的值。这样，你的程序就可以尽可能保证安全，而又不影响到用户体验。

如果你在一个Web程序中使用密钥扩展，记得你需要额外的资源处理大量认证请求，并且密钥扩展也使得网站更容易遭受拒绝服务攻击（DoS）。但我依然推荐使用密钥扩展，不过把迭代次数设定得低一点，你应该基于认证请求最高峰时的剩余硬件资源来计算迭代次数。要求用户每次登录时输入验证码可以消除拒绝服务的威胁。另外，一定要把你的系统设计为迭代次数可随时调整的。

如果你担心计算量带来的负载，但又想在Web程序中使用密钥扩展，可以考虑在浏览器中用JavaScript完成。[Stanford JavaScript Crypto Library](http://crypto.stanford.edu/sjcl/)里包含了PBKDF2的实现。迭代次数应该被设置到足够低，以适应速度较慢的客户端，比如移动设备。同时当客户端不支持JavaScript的时候，服务端应该接手计算。客户端的密钥扩展并不能免除服务端进行哈希加密的职责，你必须对客户端传来的哈希值再次进行哈希加密，就像对付一个普通密码一样。

**无法破解的哈希加密：密钥哈希和密码哈希设备**

只要攻击者可以检测对一个密码的猜测是否正确，那么他们就可以进行字典攻击或暴力攻击。因此下一步就是向哈希计算中增加一个**密钥**，只有知道这个密钥的人才能校验密码。有两种办法可以实现：将哈希值加密，比如使用AES算法；将密钥包含到哈希字符串中，比如使用密钥哈希算法[HMAC](http://en.wikipedia.org/wiki/HMAC)。

听起来很简单，做起来就不一样了。这个密钥需要在任何情况下都不被攻击者获取，即使系统因为漏洞被攻破了。如果攻击者获取了进入系统的最高权限，那么不论密钥被储存在哪，他们都可以窃取到。因此密钥需要储存在外部系统中，比如另一个用于密码校验的物理服务器，或者一个关联到服务器的特制硬件，如[YubiHSM](https://www.yubico.com/YubiHSM)。

我强烈推荐大型服务（10万用户以上）使用这类办法，因为我认为面对如此多的用户是有必要的。

如果你难以负担多个服务器或专用的硬件，仍然有办法在一个普通Web服务器上利用密钥哈希技术。大部分针对数据库的入侵都是由于[SQL注入攻击](http://en.wikipedia.org/wiki/SQL_injection)，因此不要给攻击者进入本地文件系统的权限（禁止数据库服务访问本地文件系统，如果它有这个功能的话）。这样一来，当你随机生成一个密钥存到通过Web程序无法访问的文件中，然后混入加盐哈希，得到的哈希值就不再那么脆弱了，即便这时数据库遭受了注入攻击。不要把将密钥硬编码到代码里，应该在安装时随机生成。这当然不如独立的硬件系统安全，因为如果Web程序存在SQL注入点，那么可能还存在其他一些问题，比如本地文件包含漏洞（Local File Inclusion），攻击者可以利用它读取本地密钥文件。无论如何，这个措施比没有好。

请注意密钥哈希不代表无需进行加盐。高明的攻击者迟早会找到办法窃取密钥，因此依然对密码哈希进行加盐和密钥扩展很重要。

**其他安全措施**

哈希加密可以在系统发生入侵时保护密码，但这并不能使整个程序更加安全。首先还有很多事情需要做，来保证密码哈希（和其他用户数据）不被窃取。

即使经验丰富的开发者也需要额外学习安全知识，才能写出安全的程序。这里有个关于Web程序漏洞的资源：[The Open Web Application Security Project (OWASP)](https://www.owasp.org/index.php/Main_Page)，还有一个很好的介绍：[OWASP Top Ten Vulnerability List](http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf)。除非你了解列表中所有的漏洞，才能尝试编写一个处理敏感数据的Web程序。雇主也有责任保证他所有的开发人员都有资质编写安全的程序。

对你的程序进行第三方“渗透测试”是一个不错的选择。最好的程序员也可能犯错，因此有一个安全专家审查你的代码寻找潜在的漏洞是有意义的。找寻值得信赖的机构（或招聘人员）来对你的代码进行审查。安全审查应该从编码的初期就着手进行，一直贯穿整个开发过程。

监控你的网站来发现入侵行为也是很重要的，我推荐至少雇佣一个人全职负责监测和处理安全隐患。如果有个漏洞没被发现，攻击者可能通过网站利用恶意软件感染访问者，因此检测漏洞并且及时应对是十分重要的。

## 常见问题

**我应该使用什么哈希算法？**

**应该**使用：

* 本文末尾的PHP source code, Java source code, C# source code or the Ruby source code
* OpenWall的[Portable PHP password hashing framework](http://www.openwall.com/phpass/)
* 任何先进的、被良好测试过的哈希加密算法，比如SHA256，SHA512，RipeMD，WHIRLPOOL，SHA3等等
* 设计良好的密钥扩展算法，如[PBKDF2](http://en.wikipedia.org/wiki/PBKDF2)，[bcrypt](http://en.wikipedia.org/wiki/Bcrypt)，[scrypt](http://www.tarsnap.com/scrypt.html)
* 安全的crypt()版本（$2y$，$5$，$6$）

**不要**使用：

* 过时的函数，比如MD5或SHA1
* 不安全的crypt()版本（$1$，$2$，$2x$，$3$）
* 任何你自己设计的加密算法。只应该使用那些在公开领域中的，并且被密码学家完整测试过的技术

尽管还没有一种针对MD5或SHA1非常效率的攻击手段，但是它们太古老也被广泛地认为不足以胜任存储密码的工作（某种程度上甚至是错误的），因此我也不推荐使用它们。但是有个例外，PBKDF2中频繁地使用了SHA1作为它底层的哈希函数。

**当用户忘记密码的时候，怎样进行重置？**

我个人的观点是，当前所有广泛使用的密码重置机制都是不安全的。如果你对安全性有极高的要求，比如一个加密服务，那么不要允许用户重置密码。  
大多数网站向那些忘记密码的用户发送电子邮件来进行身份认证。首先，需要随机生成一个**一次性**的令牌，它直接关联到用户的账户。然后将这个令牌混入一个重置密码的链接中，发送到用户的电子邮箱。最后当用户点击这个包含有效令牌的链接时，提示他们可以设置新的密码。要确保这个令牌只对一个账户有效，以防攻击者从邮箱获取到令牌后，用来重置其他用户的密码。

令牌必须在15分钟内使用，并且一旦被使用就立即失效。当用户重新请求令牌时，或用户登录成功时（说明他还记得密码），使原令牌失效也是一个好做法。如果一个令牌始终不过期，那么它一直可以用于入侵用户的帐号。电子邮件（SMTP）是一个纯文本协议，并且网络上有很多恶意路由在截取邮件信息。在用户修改密码后，那些包含重置密码链接的邮件在很长一段时间内依然缺乏保护。因此应该尽早使令牌过期，降低把用户信息暴露给攻击者的可能。

攻击者是可以篡改令牌的，所以不要把账户信息和失效时间存储在里面。这些信息应该以不可猜解的二进制形式存在，并且只用来识别数据库中某条用户的记录。

永远不要通过电子邮件向用户发送新密码，同时也记得在用户重置密码的时候随机生成一个新的盐值用于加密，不要重复使用之前密码的那个盐值。

**当账户数据库被泄漏或入侵时，应该怎么做？**

你首先需要做的，是查看系统被暴露到什么程度了，然后修复这个攻击者利用的漏洞。如果你没有应对入侵的经验，我强烈推荐雇一个第三方安全机构来做这件事。

将一个漏洞精心掩盖期待没有人能注意到，是否听起来很省事而又诱人呢？但是这样只会让你显得更糟糕，因为你在用户不知情的情况下，将他们的密码和个人信息暴露在危险之中。即使用户还无法理解到底发生了什么，你也应该尽快履行告知的义务。比如在首页放置一个链接，指向对此问题更详细的说明，可能的话还可以通过电子邮件告知用户目前的情况。

向你的用户说明你是如何保护他们的密码的——最好是使用了加盐哈希——即便如此恶意黑客也能使用字典攻击和暴力攻击。设想用户可能在很多服务中使用相同的密码，攻击者会用找到的密码去尝试登录其他网站。提示你的用户应该修改所有相似的密码，不论它们被使用在哪个服务上，并且强制用户下次登录你的网站时修改密码。大部分用户会尝试将密码“修改”为和之前相同的以便记忆，你应该使用老密码的哈希值来确保用户无法这么做。

即使有加盐哈希的保护，攻击者也很可能快速破解其中一些脆弱的密码。为了减少攻击者使用的它们机会，你应该对这些密码的帐号发送认证电子邮件，直到用户修改了密码。可以参考上一个问题，其中有一些实现电子邮件认证的要点。

另外也要告诉你的用户，网站到底储存了哪些个人信息。如果你的数据库中有用户的信用卡号，你应该指导用户检查自己近期的账单，并且注销掉这张信用卡。

**我应该使用什么样的密码规则？是否应该强制用户使用复杂的密码？**

如果你的服务对安全性没有严格的要求，那么不要对用户进行限制。我推荐在用户输入密码的时候，页面上显示出密码强度，由用户自己决定需要多安全的密码。如果你的服务对安全有特殊的需求，那就应该强制用户输入长度至少为12个字符的密码，并且其中至少包括两个字母、两个数字和两个符号。

不要过于频繁地强制你的用户修改密码，最多6个月1次，因为那样做会使用户疲于选择一个强度足够好的密码。更好的做法是指导用户在他们感觉密码可能泄漏的时候去主动修改，并且提示用户不要把密码告诉任何人。如果这是在商业环境中，鼓励你的员工利用工作时间熟记并使用他们的密码。

**如果攻击者入侵了我的数据库，他们难道不能把其中的密码哈希替换为自己的值，然后登录系统么？**

当然可以，但是如果他已经入侵了你的数据库，那么很可能已经有权限访问你服务器上任何东西了，因此完全没必要登录账户去获取他想要的。对密码进行哈希加密的手段，（对网站而言）不是保护网站免受入侵，而是在入侵已经发生时保护数据库中的密码。

通过为数据库连接设置两种权限，可以防止密码哈希在遭遇注入攻击时被篡改。一种权限用于创建用户：它对用户表可读可写；另一种用于用户登录，它只能读用户表而不能写。

**为什么我非得用像HMAC那种特殊的算法？为什么不能简单地把密钥混入密码？**

像MD5、SHA1和SHA2这类哈希函数是基于[Merkle–Damgård](http://en.wikipedia.org/wiki/Merkle%C3%A2%C2%80)构造的，因此在长度扩展攻击面前非常脆弱。就是说如果已经知道一个哈希值H(X)，对于任意的字符串Y，攻击者可以计算出H(pad(X) + Y)的值，而不需要知道X是多少，其中pad(X)是哈希函数的填充函数（padding function，比如MD5将数据每512bit分为一组，最后不足的将填充字节）。

在攻击者不知道密钥（key）的情况下，他仍然可以根据哈希值H(key + message)计算出H(pad(key + message) + extension)。如果这个哈希值用于身份认证，并且依靠其中的密钥来防止攻击者篡改消息，这个办法已经行不通了。因为攻击者无需知道密钥，也能构造出包含message + extension的一个有效的哈希值。

目前还不清楚攻击者能否用这个办法更快破解密码，但是由于这种攻击的出现，在密钥哈希中使用上述哈希函数已经被认为是差劲的实践了。也许某天高明的密码学家会发现一个利用长度扩展攻击的新思路，从而更快地破解密码，所以还是使用HMAC吧。

**盐值应该加到密码前面还是后面？**

都行，但是在一个程序中应该保持一致，以免出现互操作方面的问题。目前看来加到密码之前是比较常用的做法。

**为什么本文中的代码在比较哈希值的时候，都是经过固定的时间才返回结果？**

让比较过程耗费固定的时间可以保证攻击者无法对一个在线系统使用计时攻击，以此获取密码的哈希值，然后进行本地破解工作。

比较两个字节序列（字符串）的标准做法是，从第一字节开始，每个字节逐一顺序比较。只要发现某字节不相同了，就可以立即返回“假”的结果。如果遍历整个字符串也没有找到不同的字节，那么两个字符串就是相同的，并且返回“真”。这意味着比较字符串的耗时决定于两个字符串到底有多大的不同。

举个例子，使用标准的方法比较“xyzabc”和“abcxyz”，由于第一个字符就不同，不需要检查后面的内容就可以马上返回结果。相反，如果比较“aaaaaaaaaaB”和“aaaaaaaaaaZ”，比较算法就需要遍历最后一位前所有的“a”，然后才能知道它们是不相同的。

假设攻击者妄图入侵一个在线系统，并且此系统限制了每秒只能尝试一次用户认证。还假设他已经知道了密码哈希所有的参数（盐值、哈希函数的类型等等），除了密码的哈希值和密码本身（显然啊，否则还破解个什么）。如果攻击者能精确测量在线系统耗时多久去比较他猜测的密码和真实密码，那么他就能使用计时攻击获取密码的哈希值，然后进行离线破解，从而绕过系统对认证频率的限制。

首先攻击者准备256个字符串，它们的哈希值的第一字节包含了所有可能的情况。然后用它们去系统中尝试登录，并记录系统返回结果所消耗的时间，耗时最长的那个就是第一字节猜对的那个。接下来用同样的方式猜测第二字节、第三字节等等。直到攻击者获取了最够长的哈希值片段，最后只需在自己的机器上破解即可，完全不受在线系统的限制。

乍看之下在网络上进行计时攻击是不可能做到的，然而有人已经实现了，并运用到[实际中](https://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf)了。因此本文提供的代码才使用固定的时间去比较字符串，不论它们有多相似。

**“慢比较”的代码是如何工作的？**

上一个问题解释了为什么“慢比较”是有必要的，现在来讲解一下代码具体是怎么实现的。

Java



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | private static boolean slowEquals(byte[] a, byte[] b)  {      int diff = a.length ^ b.length;      for(int i = 0; i < a.length && i < b.length; i++)      diff |= a[i] ^ b[i];      return diff == 0;  } |

代码中使用了异或运算符“^”（XOR）来比较两个整数是否相等，而不是“==”。当且仅当两位相等时，异或的结果才是0。因为0 XOR 0 = 0, 1 XOR 1 = 0, 0 XOR 1 = 1, 1 XOR 0 = 1。应用到整数中每一位就是说，当且仅当字节两个整数各位都相等，结果才是0。

代码中的第一行，比较a.length和b.length，相同的话diff是0，否则diff非0。然后使用异或比较数组中各字节，并且将结果和diff求或。如果有任何一个字节不相同，diff就会变成非0的值。因为或运算没有“置0”的功能，所以循环结束后diff是0的话只有一种可能，那就是循环前两个数组长度相等（a.length == b.length），并且数组中每一个字节都相同（每次异或的结果都非0）。  
我们使用XOR而不是“==”来比较整数的原因是：“==”通常被翻译/编译/解释为带有分支的语句。例如C语言中的“diff &= a == b”可能在x86机器成被编译为如下汇编语言：

MOV EAX, [A]  
CMP [B], EAX  
JZ equal  
JMP done  
equal:  
AND [VALID], 1  
done:  
AND [VALID], 0

其中的分支导致代码运行的时间不固定，决定于两个整数相等的程度和CPU内部的跳转预测机制（branch prediction）。

而C语言代码“diff |=a ^ b”会被编译为下面的样子，它执行的时间和两个整数是什么样的情况无关。

MOV EAX, [A]  
XOR EAX, [B]  
OR [DIFF], EAX

**弄这么麻烦干嘛？**

用户在你的网站上输入密码，说明他们相信你会保障密码的安全。如果你的数据库被黑了，又没有对用户密码加以保护，恶意黑客就可以使用这些密码去入侵用户在其他网站或服务的账户（大部分人会在各处使用相同的密码）。这不仅仅关乎你网站的安全，更关系到用户的。你需要对用户的安全负责。

## PHP PBKDF2 密码哈希代码

* [PHP Source Code](http://blog.jobbole.com/61872/#php)
* [Java Source Code](http://blog.jobbole.com/61872/#java)
* [ASP.NET (C#) Source Code](http://blog.jobbole.com/61872/#csharp)
* [Ruby (on Rails) Source Code](http://blog.jobbole.com/61872/#ruby)

下面是PBKDF2在PHP中一种安全的实现，你也可以在[这个页面](https://defuse.ca/php-pbkdf2.htm)找到测试用例和基准测试的代码。

[**下载PasswordHash.php**](https://crackstation.net/source/password-hashing/PasswordHash.php)

如果你需要兼容的PHP和C＃代码，点击[这里](https://github.com/defuse/password-hashing/tree/master/compatible)。

PHP



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139 | <?php  /\*  \* Password Hashing With PBKDF2 (http://crackstation.net/hashing-security.htm).  \* Copyright (c) 2013, Taylor Hornby  \* All rights reserved.  \*  \* Redistribution and use in source and binary forms, with or without  \* modification, are permitted provided that the following conditions are met:  \*  \* 1. Redistributions of source code must retain the above copyright notice,  \* this list of conditions and the following disclaimer.  \*  \* 2. Redistributions in binary form must reproduce the above copyright notice,  \* this list of conditions and the following disclaimer in the documentation  \* and/or other materials provided with the distribution.  \*  \* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"  \* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE  \* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE  \* ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE  \* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR  \* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF  \* SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS  \* INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN  \* CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)  \* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE  \* POSSIBILITY OF SUCH DAMAGE.  \*/    // These constants may be changed without breaking existing hashes.  define("PBKDF2\_HASH\_ALGORITHM", "sha256");  define("PBKDF2\_ITERATIONS", 1000);  define("PBKDF2\_SALT\_BYTE\_SIZE", 24);  define("PBKDF2\_HASH\_BYTE\_SIZE", 24);    define("HASH\_SECTIONS", 4);  define("HASH\_ALGORITHM\_INDEX", 0);  define("HASH\_ITERATION\_INDEX", 1);  define("HASH\_SALT\_INDEX", 2);  define("HASH\_PBKDF2\_INDEX", 3);    function create\_hash($password)  {      // format: algorithm:iterations:salt:hash      $salt = base64\_encode(mcrypt\_create\_iv(PBKDF2\_SALT\_BYTE\_SIZE, MCRYPT\_DEV\_URANDOM));      return PBKDF2\_HASH\_ALGORITHM . ":" . PBKDF2\_ITERATIONS . ":" .  $salt . ":" .          base64\_encode(pbkdf2(              PBKDF2\_HASH\_ALGORITHM,              $password,              $salt,              PBKDF2\_ITERATIONS,              PBKDF2\_HASH\_BYTE\_SIZE,              true          ));  }    function validate\_password($password, $correct\_hash)  {      $params = explode(":", $correct\_hash);      if(count($params) < HASH\_SECTIONS)         return false;      $pbkdf2 = base64\_decode($params[HASH\_PBKDF2\_INDEX]);      return slow\_equals(          $pbkdf2,          pbkdf2(              $params[HASH\_ALGORITHM\_INDEX],              $password,              $params[HASH\_SALT\_INDEX],              (int)$params[HASH\_ITERATION\_INDEX],              strlen($pbkdf2),              true          )      );  }    // Compares two strings $a and $b in length-constant time.  function slow\_equals($a, $b)  {      $diff = strlen($a) ^ strlen($b);      for($i = 0; $i < strlen($a) && $i < strlen($b); $i++)      {          $diff |= ord($a[$i]) ^ ord($b[$i]);      }      return $diff === 0;  }    /\*  \* PBKDF2 key derivation function as defined by RSA's PKCS #5: https://www.ietf.org/rfc/rfc2898.txt  \* $algorithm - The hash algorithm to use. Recommended: SHA256  \* $password - The password.  \* $salt - A salt that is unique to the password.  \* $count - Iteration count. Higher is better, but slower. Recommended: At least 1000.  \* $key\_length - The length of the derived key in bytes.  \* $raw\_output - If true, the key is returned in raw binary format. Hex encoded otherwise.  \* Returns: A $key\_length-byte key derived from the password and salt.  \*  \* Test vectors can be found here: https://www.ietf.org/rfc/rfc6070.txt  \*  \* This implementation of PBKDF2 was originally created by https://defuse.ca  \* With improvements by http://www.variations-of-shadow.com  \*/  function pbkdf2($algorithm, $password, $salt, $count, $key\_length, $raw\_output = false)  {      $algorithm = strtolower($algorithm);      if(!in\_array($algorithm, hash\_algos(), true))          trigger\_error('PBKDF2 ERROR: Invalid hash algorithm.', E\_USER\_ERROR);      if($count <= 0 || $key\_length <= 0)          trigger\_error('PBKDF2 ERROR: Invalid parameters.', E\_USER\_ERROR);        if (function\_exists("hash\_pbkdf2")) {          // The output length is in NIBBLES (4-bits) if $raw\_output is false!          if (!$raw\_output) {              $key\_length = $key\_length \* 2;          }          return hash\_pbkdf2($algorithm, $password, $salt, $count, $key\_length, $raw\_output);      }        $hash\_length = strlen(hash($algorithm, "", true));      $block\_count = ceil($key\_length / $hash\_length);        $output = "";      for($i = 1; $i <= $block\_count; $i++) {          // $i encoded as 4 bytes, big endian.          $last = $salt . pack("N", $i);          // first iteration          $last = $xorsum = hash\_hmac($algorithm, $last, $password, true);          // perform the other $count - 1 iterations          for ($j = 1; $j < $count; $j++) {              $xorsum ^= ($last = hash\_hmac($algorithm, $last, $password, true));          }          $output .= $xorsum;      }        if($raw\_output)          return substr($output, 0, $key\_length);      else          return bin2hex(substr($output, 0, $key\_length));  }  ?> |

## Java PBKDF2 密码哈希代码

* [PHP Source Code](http://blog.jobbole.com/61872/#php)
* [Java Source Code](http://blog.jobbole.com/61872/#java)
* [ASP.NET (C#) Source Code](http://blog.jobbole.com/61872/#csharp)
* [Ruby (on Rails) Source Code](http://blog.jobbole.com/61872/#ruby)

下面是PBKDF2在Java中一种安全的实现。

[**下载PasswordHash.java**](https://crackstation.net/source/password-hashing/PasswordHash.java)

Java



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235 | /\*\*  \* Password Hashing With PBKDF2 (http://crackstation.net/hashing-security.htm).  \* Copyright (c) 2013, Taylor Hornby  \* All rights reserved.  \*  \* Redistribution and use in source and binary forms, with or without  \* modification, are permitted provided that the following conditions are met:  \*  \* 1. Redistributions of source code must retain the above copyright notice,  \* this list of conditions and the following disclaimer.  \*  \* 2. Redistributions in binary form must reproduce the above copyright notice,  \* this list of conditions and the following disclaimer in the documentation  \* and/or other materials provided with the distribution.  \*  \* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"  \* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE  \* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE  \* ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE  \* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR  \* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF  \* SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS  \* INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN  \* CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)  \* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE  \* POSSIBILITY OF SUCH DAMAGE.  \*/    import java.security.SecureRandom;  import javax.crypto.spec.PBEKeySpec;  import javax.crypto.SecretKeyFactory;  import java.math.BigInteger;  import java.security.NoSuchAlgorithmException;  import java.security.spec.InvalidKeySpecException;    /\*\*  \* PBKDF2 salted password hashing.  \* Author: havoc AT defuse.ca  \* www: http://crackstation.net/hashing-security.htm  \*/  public class PasswordHash  {      public static final String PBKDF2\_ALGORITHM = "PBKDF2WithHmacSHA1";        // The following constants may be changed without breaking existing hashes.      public static final int SALT\_BYTE\_SIZE = 24;      public static final int HASH\_BYTE\_SIZE = 24;      public static final int PBKDF2\_ITERATIONS = 1000;        public static final int ITERATION\_INDEX = 0;      public static final int SALT\_INDEX = 1;      public static final int PBKDF2\_INDEX = 2;        /\*\*       \* Returns a salted PBKDF2 hash of the password.       \*       \* @param   password    the password to hash       \* @return              a salted PBKDF2 hash of the password       \*/      public static String createHash(String password)          throws NoSuchAlgorithmException, InvalidKeySpecException      {          return createHash(password.toCharArray());      }        /\*\*       \* Returns a salted PBKDF2 hash of the password.       \*       \* @param   password    the password to hash       \* @return              a salted PBKDF2 hash of the password       \*/      public static String createHash(char[] password)          throws NoSuchAlgorithmException, InvalidKeySpecException      {          // Generate a random salt          SecureRandom random = new SecureRandom();          byte[] salt = new byte[SALT\_BYTE\_SIZE];          random.nextBytes(salt);            // Hash the password          byte[] hash = pbkdf2(password, salt, PBKDF2\_ITERATIONS, HASH\_BYTE\_SIZE);          // format iterations:salt:hash          return PBKDF2\_ITERATIONS + ":" + toHex(salt) + ":" +  toHex(hash);      }        /\*\*       \* Validates a password using a hash.       \*       \* @param   password        the password to check       \* @param   correctHash     the hash of the valid password       \* @return                  true if the password is correct, false if not       \*/      public static boolean validatePassword(String password, String correctHash)          throws NoSuchAlgorithmException, InvalidKeySpecException      {          return validatePassword(password.toCharArray(), correctHash);      }        /\*\*       \* Validates a password using a hash.       \*       \* @param   password        the password to check       \* @param   correctHash     the hash of the valid password       \* @return                  true if the password is correct, false if not       \*/      public static boolean validatePassword(char[] password, String correctHash)          throws NoSuchAlgorithmException, InvalidKeySpecException      {          // Decode the hash into its parameters          String[] params = correctHash.split(":");          int iterations = Integer.parseInt(params[ITERATION\_INDEX]);          byte[] salt = fromHex(params[SALT\_INDEX]);          byte[] hash = fromHex(params[PBKDF2\_INDEX]);          // Compute the hash of the provided password, using the same salt,          // iteration count, and hash length          byte[] testHash = pbkdf2(password, salt, iterations, hash.length);          // Compare the hashes in constant time. The password is correct if          // both hashes match.          return slowEquals(hash, testHash);      }        /\*\*       \* Compares two byte arrays in length-constant time. This comparison method       \* is used so that password hashes cannot be extracted from an on-line       \* system using a timing attack and then attacked off-line.       \*       \* @param   a       the first byte array       \* @param   b       the second byte array       \* @return          true if both byte arrays are the same, false if not       \*/      private static boolean slowEquals(byte[] a, byte[] b)      {          int diff = a.length ^ b.length;          for(int i = 0; i < a.length && i < b.length; i++)              diff |= a[i] ^ b[i];          return diff == 0;      }        /\*\*       \*  Computes the PBKDF2 hash of a password.       \*       \* @param   password    the password to hash.       \* @param   salt        the salt       \* @param   iterations  the iteration count (slowness factor)       \* @param   bytes       the length of the hash to compute in bytes       \* @return              the PBDKF2 hash of the password       \*/      private static byte[] pbkdf2(char[] password, byte[] salt, int iterations, int bytes)          throws NoSuchAlgorithmException, InvalidKeySpecException      {          PBEKeySpec spec = new PBEKeySpec(password, salt, iterations, bytes \* 8);          SecretKeyFactory skf = SecretKeyFactory.getInstance(PBKDF2\_ALGORITHM);          return skf.generateSecret(spec).getEncoded();      }        /\*\*       \* Converts a string of hexadecimal characters into a byte array.       \*       \* @param   hex         the hex string       \* @return              the hex string decoded into a byte array       \*/      private static byte[] fromHex(String hex)      {          byte[] binary = new byte[hex.length() / 2];          for(int i = 0; i < binary.length; i++)          {              binary[i] = (byte)Integer.parseInt(hex.substring(2\*i, 2\*i+2), 16);          }          return binary;      }        /\*\*       \* Converts a byte array into a hexadecimal string.       \*       \* @param   array       the byte array to convert       \* @return              a length\*2 character string encoding the byte array       \*/      private static String toHex(byte[] array)      {          BigInteger bi = new BigInteger(1, array);          String hex = bi.toString(16);          int paddingLength = (array.length \* 2) - hex.length();          if(paddingLength > 0)              return String.format("%0" + paddingLength + "d", 0) + hex;          else              return hex;      }        /\*\*       \* Tests the basic functionality of the PasswordHash class       \*       \* @param   args        ignored       \*/      public static void main(String[] args)      {          try          {              // Print out 10 hashes              for(int i = 0; i < 10; i++)                  System.out.println(PasswordHash.createHash("p\r\nassw0Rd!"));                // Test password validation              boolean failure = false;              System.out.println("Running tests...");              for(int i = 0; i < 100; i++)              {                  String password = ""+i;                  String hash = createHash(password);                  String secondHash = createHash(password);                  if(hash.equals(secondHash)) {                      System.out.println("FAILURE: TWO HASHES ARE EQUAL!");                      failure = true;                  }                  String wrongPassword = ""+(i+1);                  if(validatePassword(wrongPassword, hash)) {                      System.out.println("FAILURE: WRONG PASSWORD ACCEPTED!");                      failure = true;                  }                  if(!validatePassword(password, hash)) {                      System.out.println("FAILURE: GOOD PASSWORD NOT ACCEPTED!");                      failure = true;                  }              }              if(failure)                  System.out.println("TESTS FAILED!");              else                  System.out.println("TESTS PASSED!");          }          catch(Exception ex)          {              System.out.println("ERROR: " + ex);          }      }    } |

## ASP.NET(C#) PBKDF2 密码哈希代码

* [PHP Source Code](http://blog.jobbole.com/61872/#php)
* [Java Source Code](http://blog.jobbole.com/61872/#java)
* [ASP.NET (C#) Source Code](http://blog.jobbole.com/61872/#csharp)
* [Ruby (on Rails) Source Code](http://blog.jobbole.com/61872/#ruby)

下面是PBKDF2在ASP.NET(C#)中一种安全的实现。

[**下载PasswordHash.cs**](https://crackstation.net/source/password-hashing/PasswordHash.cs)

如果你需要兼容的PHP和C＃代码，点击[这里](https://github.com/defuse/password-hashing/tree/master/compatible)。

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121 | /\*  \* Password Hashing With PBKDF2 (http://crackstation.net/hashing-security.htm).  \* Copyright (c) 2013, Taylor Hornby  \* All rights reserved.  \*  \* Redistribution and use in source and binary forms, with or without  \* modification, are permitted provided that the following conditions are met:  \*  \* 1. Redistributions of source code must retain the above copyright notice,  \* this list of conditions and the following disclaimer.  \*  \* 2. Redistributions in binary form must reproduce the above copyright notice,  \* this list of conditions and the following disclaimer in the documentation  \* and/or other materials provided with the distribution.  \*  \* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"  \* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE  \* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE  \* ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE  \* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR  \* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF  \* SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS  \* INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN  \* CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)  \* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE  \* POSSIBILITY OF SUCH DAMAGE.  \*/    using System;  using System.Text;  using System.Security.Cryptography;    namespace PasswordHash  {      /// <summary>      /// Salted password hashing with PBKDF2-SHA1.      /// Author: havoc AT defuse.ca      /// www: http://crackstation.net/hashing-security.htm      /// Compatibility: .NET 3.0 and later.      /// </summary>      public class PasswordHash      {          // The following constants may be changed without breaking existing hashes.          public const int SALT\_BYTE\_SIZE = 24;          public const int HASH\_BYTE\_SIZE = 24;          public const int PBKDF2\_ITERATIONS = 1000;            public const int ITERATION\_INDEX = 0;          public const int SALT\_INDEX = 1;          public const int PBKDF2\_INDEX = 2;            /// <summary>          /// Creates a salted PBKDF2 hash of the password.          /// </summary>          /// <param name="password">The password to hash.</param>          /// <returns>The hash of the password.</returns>          public static string CreateHash(string password)          {              // Generate a random salt              RNGCryptoServiceProvider csprng = new RNGCryptoServiceProvider();              byte[] salt = new byte[SALT\_BYTE\_SIZE];              csprng.GetBytes(salt);                // Hash the password and encode the parameters              byte[] hash = PBKDF2(password, salt, PBKDF2\_ITERATIONS, HASH\_BYTE\_SIZE);              return PBKDF2\_ITERATIONS + ":" +                  Convert.ToBase64String(salt) + ":" +                  Convert.ToBase64String(hash);          }            /// <summary>          /// Validates a password given a hash of the correct one.          /// </summary>          /// <param name="password">The password to check.</param>          /// <param name="correctHash">A hash of the correct password.</param>          /// <returns>True if the password is correct. False otherwise.</returns>          public static bool ValidatePassword(string password, string correctHash)          {              // Extract the parameters from the hash              char[] delimiter = { ':' };              string[] split = correctHash.Split(delimiter);              int iterations = Int32.Parse(split[ITERATION\_INDEX]);              byte[] salt = Convert.FromBase64String(split[SALT\_INDEX]);              byte[] hash = Convert.FromBase64String(split[PBKDF2\_INDEX]);                byte[] testHash = PBKDF2(password, salt, iterations, hash.Length);              return SlowEquals(hash, testHash);          }            /// <summary>          /// Compares two byte arrays in length-constant time. This comparison          /// method is used so that password hashes cannot be extracted from          /// on-line systems using a timing attack and then attacked off-line.          /// </summary>          /// <param name="a">The first byte array.</param>          /// <param name="b">The second byte array.</param>          /// <returns>True if both byte arrays are equal. False otherwise.</returns>          private static bool SlowEquals(byte[] a, byte[] b)          {              uint diff = (uint)a.Length ^ (uint)b.Length;              for (int i = 0; i < a.Length && i < b.Length; i++)                  diff |= (uint)(a[i] ^ b[i]);              return diff == 0;          }            /// <summary>          /// Computes the PBKDF2-SHA1 hash of a password.          /// </summary>          /// <param name="password">The password to hash.</param>          /// <param name="salt">The salt.</param>          /// <param name="iterations">The PBKDF2 iteration count.</param>          /// <param name="outputBytes">The length of the hash to generate, in bytes.</param>          /// <returns>A hash of the password.</returns>          private static byte[] PBKDF2(string password, byte[] salt, int iterations, int outputBytes)          {              Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(password, salt);              pbkdf2.IterationCount = iterations;              return pbkdf2.GetBytes(outputBytes);          }      }  } |

## Ruby(on Rails) PBKDF2 密码哈希代码

* [PHP Source Code](http://blog.jobbole.com/61872/#php)
* [Java Source Code](http://blog.jobbole.com/61872/#java)
* [ASP.NET (C#) Source Code](http://blog.jobbole.com/61872/#csharp)
* [Ruby (on Rails) Source Code](http://blog.jobbole.com/61872/#ruby)

下面是PBKDF2在Ruby(on Rails)中一种安全的实现。

[**下载PasswordHash.rb**](https://crackstation.net/source/password-hashing/PasswordHash.rb)



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117 | # Password Hashing With PBKDF2 (http://crackstation.net/hashing-security.htm).  # Copyright (c) 2013, Taylor Hornby  # All rights reserved.  #  # Redistribution and use in source and binary forms, with or without  # modification, are permitted provided that the following conditions are met:  #  # 1. Redistributions of source code must retain the above copyright notice,  # this list of conditions and the following disclaimer.  #  # 2. Redistributions in binary form must reproduce the above copyright notice,  # this list of conditions and the following disclaimer in the documentation  # and/or other materials provided with the distribution.  #  # THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS &quot;AS IS&quot;  # AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE  # IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE  # ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE  # LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR  # CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF  # SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS  # INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN  # CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)  # ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE  # POSSIBILITY OF SUCH DAMAGE.    require &#039;securerandom&#039;  require &#039;openssl&#039;  require &#039;base64&#039;    # Salted password hashing with PBKDF2-SHA1.  # Authors: @RedragonX (dicesoft.net), havoc AT defuse.ca  # www: http://crackstation.net/hashing-security.htm  module PasswordHash      # The following constants can be changed without breaking existing hashes.    PBKDF2\_ITERATIONS = 1000    SALT\_BYTE\_SIZE = 24    HASH\_BYTE\_SIZE = 24      HASH\_SECTIONS = 4    SECTION\_DELIMITER = &#039;:&#039;    ITERATIONS\_INDEX = 1    SALT\_INDEX = 2    HASH\_INDEX = 3      # Returns a salted PBKDF2 hash of the password.    def self.createHash( password )      salt = SecureRandom.base64( SALT\_BYTE\_SIZE )      pbkdf2 = OpenSSL::PKCS5::pbkdf2\_hmac\_sha1(        password,        salt,        PBKDF2\_ITERATIONS,        HASH\_BYTE\_SIZE      )      return [&quot;sha1&quot;, PBKDF2\_ITERATIONS, salt, Base64.encode64( pbkdf2 )].join( SECTION\_DELIMITER )    end      # Checks if a password is correct given a hash of the correct one.    # correctHash must be a hash string generated with createHash.    def self.validatePassword( password, correctHash )      params = correctHash.split( SECTION\_DELIMITER )      return false if params.length != HASH\_SECTIONS        pbkdf2 = Base64.decode64( params[HASH\_INDEX] )      testHash = OpenSSL::PKCS5::pbkdf2\_hmac\_sha1(        password,        params[SALT\_INDEX],        params[ITERATIONS\_INDEX].to\_i,        pbkdf2.length      )        return pbkdf2 == testHash    end      # Run tests to ensure the module is functioning properly.    # Returns true if all tests succeed, false if not.    def self.runSelfTests      puts &quot;Sample hashes:&quot;      3.times { puts createHash(&quot;password&quot;) }        puts &quot;\nRunning self tests...&quot;      @@allPass = true        correctPassword = &#039;aaaaaaaaaa&#039;      wrongPassword = &#039;aaaaaaaaab&#039;      hash = createHash(correctPassword)        assert( validatePassword( correctPassword, hash ) == true, &quot;correct password&quot; )      assert( validatePassword( wrongPassword, hash ) == false, &quot;wrong password&quot; )        h1 = hash.split( SECTION\_DELIMITER )      h2 = createHash( correctPassword ).split( SECTION\_DELIMITER )      assert( h1[HASH\_INDEX] != h2[HASH\_INDEX], &quot;different hashes&quot; )      assert( h1[SALT\_INDEX] != h2[SALT\_INDEX], &quot;different salt&quot; )        if @@allPass        puts &quot;\*\*\* ALL TESTS PASS \*\*\*&quot;      else        puts &quot;\*\*\* FAILURES \*\*\*&quot;      end        return @@allPass    end      def self.assert( truth, msg )      if truth        puts &quot;PASS [#{msg}]&quot;      else        puts &quot;FAIL [#{msg}]&quot;        @@allPass = false      end    end    end    PasswordHash.runSelfTests |

文章和代码由[Defuse Security](https://defuse.ca/)编写。