Keep your PHP web applications' passwords safe from hackers. PHP continues to evolve with the web, and more web applications need to store passwords securely. In the second installment of the four-part PHP renewed series, learn about the new password-handling features added in the PHP 5.5 release.

View more content in this series

PHP, from the beginning, was a programming language made for building websites. That idea is in PHP's core far deeper than in any other programming language — perhaps one reason why PHP became and remains so popular for building web applications. But when PHP was first crafted in the mid-1990s, the term web application didn't even exist yet. Password protection, then, wasn't one of the features that the PHP creators devoted resources to. After all, you didn't need to worry about passwords when you used PHP just to put a site-visit counter or a date-modified stamp on your web page.

But 20 years have passed, and now it's almost unthinkable to create a web application that doesn't involve password-protected user accounts. It's of the utmost importance that PHP programmers safeguard account passwords by using the latest and most secure methods. To that end, PHP 5.5 added a new password-hashing library created by Anthony Ferrara (@ircmaxell). The library makes several functions available that you can use to handle one-way password encryption with current best-practice methods. Other features anticipate future security needs so that as computers and hackers get more advanced, you can stay a step ahead of the bad guys. This article gives you an in-depth introduction to the library's functions and how to make the best use of them.

About this series

Continuously evolving under the aegis of an active open source project, PHP now powers much of the web. PHP has undergone remarkable changes since its early days as a templating language. If you haven't used or evaluated PHP technology in many years, you'd barely even recognize some current PHP projects. This series shows you the latest PHP features and how to use today's PHP to build modern, secure websites.
Importance of secure hashes

“Treat your password like your toothbrush. Don't let anybody else use it, and get a new one every six months.”
Clifford Stoll

This probably goes without saying (at least I hope it does): Never store a user's password as plain text. Always store an encrypted version of the password by using a one-way encryption algorithm such as a hashing algorithm to make it impossible for anyone with access to your account database to discover what your users' passwords are. This admonition isn't only for protecting your users from someone who gains access to the database by compromising your website. You also need to protect against people within your own organization who might have malicious intent.

What is a hash?

A hashing algorithm is one that takes a string of any length and consistently creates a hash: a fixed-size string representation. Every time you pass in the same original string, you receive back the same hash. This is a one-way process from which you cannot get the original string back.

This need is all the greater because many users, unfortunately, use identical passwords for multiple websites. If someone gets access to the email address and raw password of one of your users, chances are good that they'll be able to access that user's accounts on other websites.

Cracking speed

All hashes are not created equal. You can use various algorithms to create a hash. Two that were used commonly in the past are MD5 and SHA-1. Today's powerful computers can fairly easily crack both of these algorithms. For example, software exists to crack hashes at the rate of 3,650 million per second against MD5, and 1,360 million per second against SHA-1 on a single GPU. At those rates, depending on the complexity and length of a password, it's possible to crack one in less than an hour.

So, it's important to use more computationally complex hashing algorithms. Not only do you want a longer hash (which reduces the chance of hash collisions — two phrases generating the same hash), but you also want it to take as long as possible for the hash to be generated. Why? For your own web application, users need to wait for a password hash to be generated only once each time they log in. If that wait lasts a second (or even a couple of seconds), the user won't care or even notice. But by slowing a cracking attempt from 3.6 billion calculations per second down to 1 per second, you make any attempt exponentially harder.
Rainbow tables

You also need to protect yourself against rainbow tables. Rainbow tables are reverse-lookup tables for hashes. The table’s creator precalculates the MD5 hashes for all common words, phrases, modified words, and even random strings. Someone with access to a hash can enter it into a lookup to discover the password that was used to generate it, effectively reversing the one-way process. The relatively low processing cost of cracking an MD5 hash makes the creation of this rainbow table possible.

Generating a rainbow table for a computationally complex algorithm takes much longer. But it's still possible and only a one-time pain for the creator. The appropriate countermeasure is to add a salt to your hash. In this context, salt refers to any phrase that's added to your password before you create the hash in the first place. By using a salt you end up defeating (in practice) rainbow tables. Someone would need to generate a rainbow table specific to your application and then, from cracking multiple passwords, figure out what your salt is — a difficult and expensive scenario.

Improving on past PHP password practices

Now take a look at Listing 1, an example of what good password practice in PHP looked like several years ago.

Listing 1. What used to be considered good password security in PHP

```php
<?php
// Create a password class to handle management of this:
class Password {
    const SALT = 'MyVoiceIsMyPassport';

    public static function hash($password) {
        return hash('sha512', self::SALT . $password);
    }

    public static function verify($password, $hash) {
        return ($hash == self::hash($password));
    }
}

// Hash the password:
$hash = Password::hash('correct horse battery staple');

// Check against an entered password (This example will fail to verify)
if (Password::verify('Tr0ub4dor&3', $hash)) {
    echo 'Correct Password!\n';
} else {
    echo "Incorrect login attempt!\n";
}
```

Examples such as Listing 1 litter the web as so-called best practices. And for a long time, this approach was the best practice — certainly better than using MD5, and drastically better than storing passwords as plain text. Listing 1 uses the much more complex SHA-512 algorithm, and it forces all passwords to be salted to defeat premade rainbow tables. But a few issues remain with this approach.
Moving to a random salt

Listing 1 uses a salt, but every password uses the exact same salt. So, once someone cracks a single password (or worse perhaps, discovers the salt by getting access to the codebase), a custom rainbow table can be made by adding the salt to each rainbow table entry. The rainbow-table-defeating solution is to use a random salt for each password as it is created, and to store the salt along with the password in such a way that it can be retrieved.

Further increasing the cost

Listing 1 also uses SHA-512 — a much more complex algorithm that comes with PHP — instead of MD5 or SHA-1. However, even SHA-512 hashes can be cracked at the rate of around 46 million calculations per second. Although slower than the MD5 or SHA1 rates, this rate still isn’t nearly slow enough for adequate security. The solution to this problem is to use algorithms that are even more computationally complex, and to run those algorithms multiple times. For example, running every password through SHA-512 100 separate times consecutively would slow down any hack attempt considerably.

The good news is that you needn't try to implement this solution with your own code. The new password-hashing library in PHP 5.5 addresses that very need.

Introducing password_hash()

The password-hashing extension creates computationally complex secure password hashes for you, including generating and handling random salts behind the scenes. In the simplest use case you call password_hash() with the password that you want a hash for, and the extension handles everything for you. You also need to provide a second parameter: the hash algorithm that you want the extension to use. You have a couple of choices, but specifying the PASSWORD_DEFAULT constant is the best option at the moment (I'll explain why in a minute):

```php
<?php
$hash = password_hash('correct horse battery staple', PASSWORD_DEFAULT);
```

Specifying a cost

You can also provide a third parameter, which is an array of options that change how the hash is generated. You can specify the salt there, but it's best if you don't and allow a random salt to be generated for you. More important, in this array you can specify a cost value. This value — which defaults to 10 — determines how complex the algorithm should be and therefore how long generating the hash will take. (Consider this value as changing the number of times the algorithm is rerun on top of itself to slow down the calculations.) If you want a more secure password and your hardware is capable of handing it, you might make your call like this:

```php
<?php
$hash = password_hash('correct horse battery staple', PASSWORD_DEFAULT, ['cost' => 14]);
```

Using my own MacBook Pro as a test environment, generating a password_hash with cost 10 (the default) takes around 0.085 seconds. Upping that cost to 14 changes that time to a serious 1.394 seconds per calculation.
Verifying generated passwords

Because the passwords in the preceding two examples were generated with a random salt process for me, I don't directly know the salt in question. So if I try to run `password_hash()` again and compare the strings as a way to verify the password, the results won't match. Every time you call the function, a new salt is generated and the returned hash is different. So the extension provides a second function, called `password_verify()`, that handles the verification process for you. You call `password_verify()`, passing in the password provided by the user and the stored hash, and the function returns a Boolean `TRUE` if the password is correct or `FALSE` if it's incorrect:

```php
define_string
if (password_verify($password, $hash)) {
    // Correct Password
}
define_string```

Now I can refactor the class in Listing 1 to use the built-in password-hashing extension, as shown in Listing 2.

Listing 2. Refactoring of Listing 1's Password class

```php
define_string
<?php
define_string
class Password {
    public static function hash($password) {
        return password_hash($password, PASSWORD_DEFAULT, ['cost' => 14]);
    }

define_string
    public static function verify($password, $hash) {
        return password_verify($password, $hash);
    }
}
define_string
```  

Handling changing security needs

By using the new password-hashing extension, you can bring your codebase up to today's security standards. But only a few years ago experts were saying that SHA-1 was a best-practice solution. So what happens if — no, when — password encryption needs to get stronger? Luckily, the new extension has a built-in feature that takes this eventuality into account.

The `password_needs_rehash()` function can be used to detect — behind the scenes — if the stored password doesn't match the current security needs you've specified. The reason might be that you've increased the `cost` parameter, or that a new PHP release changes behind the scenes to a different hash algorithm. This is why `PASSWORD_DEFAULT` should be your algorithm choice; that option will always cause your software to use the current best-practice version.

Using this feature is a little more complicated but not overly so. When checking against a user's password (such as when the user tries to log in), you perform one extra task: a call to `password_needs_rehash()`, which takes similar parameters to `password_hash()`. The `password_needs_rehash()` function checks the provided password hash against the newly requested security settings. If the password hash doesn't match those settings, the function reports that fact back to you.
Some programmers get confused here, because all that the password_needs_rehash() function does is let you know if the password needs a rehash. It's completely up to you to then generate a new hash of the password and save it — because the password extension doesn't know how you might need to store the password.

In Listing 3, I present a full mockup User class in which, using the tools I've discussed, I both handle a user's password securely and allow for future changing security needs.

**Listing 3. Mock User class showing full use of the password extension**

```php
<?php

class User {
    // Store password options so that rehash & hash can share them:
    const HASH = PASSWORD_DEFAULT;
    const COST = 14;

    // Internal data storage about the user:
    public $data;

    // Mock constructor:
    private function __construct() {
        // Read data from the database, storing it into $data such as:
        // $data->passwordHash and $data->username
        $this->data = new stdClass();
        $this->data->passwordHash = 'dbd814125a4bad51db85f27279f1040a';
    }

    // Mock save functionality
    public function save() {
        // Store the data from $data back into the database
    }

    // Allow for changing a new password:
    public function setPassword($password) {
        $this->data->passwordHash = password_hash($password, self::HASH, ['cost' => self::COST]);
    }

    // Logic for logging a user in:
    public function login($password) {
        // First see if they gave the right password:
        echo "Login: ", $this->data->passwordHash, "\n";
        if (password_verify($password, $this->data->passwordHash)) {
            // Success - Now see if their password needs rehashed
            if (password_needs_rehash($this->data->passwordHash, self::HASH, ['cost' => self::COST])) {
                // We need to rehash the password, and save it. Just call setPassword
                $this->setPassword($password);
                $this->save();
            }
            return true; // Or do what you need to mark the user as logged in.
        }
        return false;
    }
}
```

**Conclusion**

Now you know how the new password-hashing library works for PHP and how it will continue to help protect your users against security breaches. In the next *PHP renewed* installment, I'll shift the discussion from the evolution of the PHP language itself to the ecosystem and tools that have started to evolve around it — beginning with Composer, a dependency manager for PHP that has taken the community by storm.
Related topics

- hash() documentation on php.net
- Password hashing documentation on php.net
- PHP project resources: Check out the developerWorks PHP project resources to expand your PHP skills.
- The Rainbow Table Is Dead (ircmaxell's blog, August 2011): Read why Anthony Ferrara considers brute forcing to be a bigger threat than rainbow tables.
- PHP: The Right Way: Learn more about how to build PHP projects in a modern fashion, including the use of Password Hashing.
- developerWorks Premium: Provides an all-access pass to powerful tools, curated technical library from Safari Books Online, conference discounts and proceedings, SoftLayer and Bluemix credits, and more.
- PHP Documentation: Consult the official source of all PHP documentation.
- PHPDeveloper.org: Get news, views, and community information for PHP.
- php[architect]: Check out an online and print magazine dedicated to PHP education and recent news.
- More PHP content: Browse all the PHP content on developerWorks.

© Copyright IBM Corporation 2015
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)