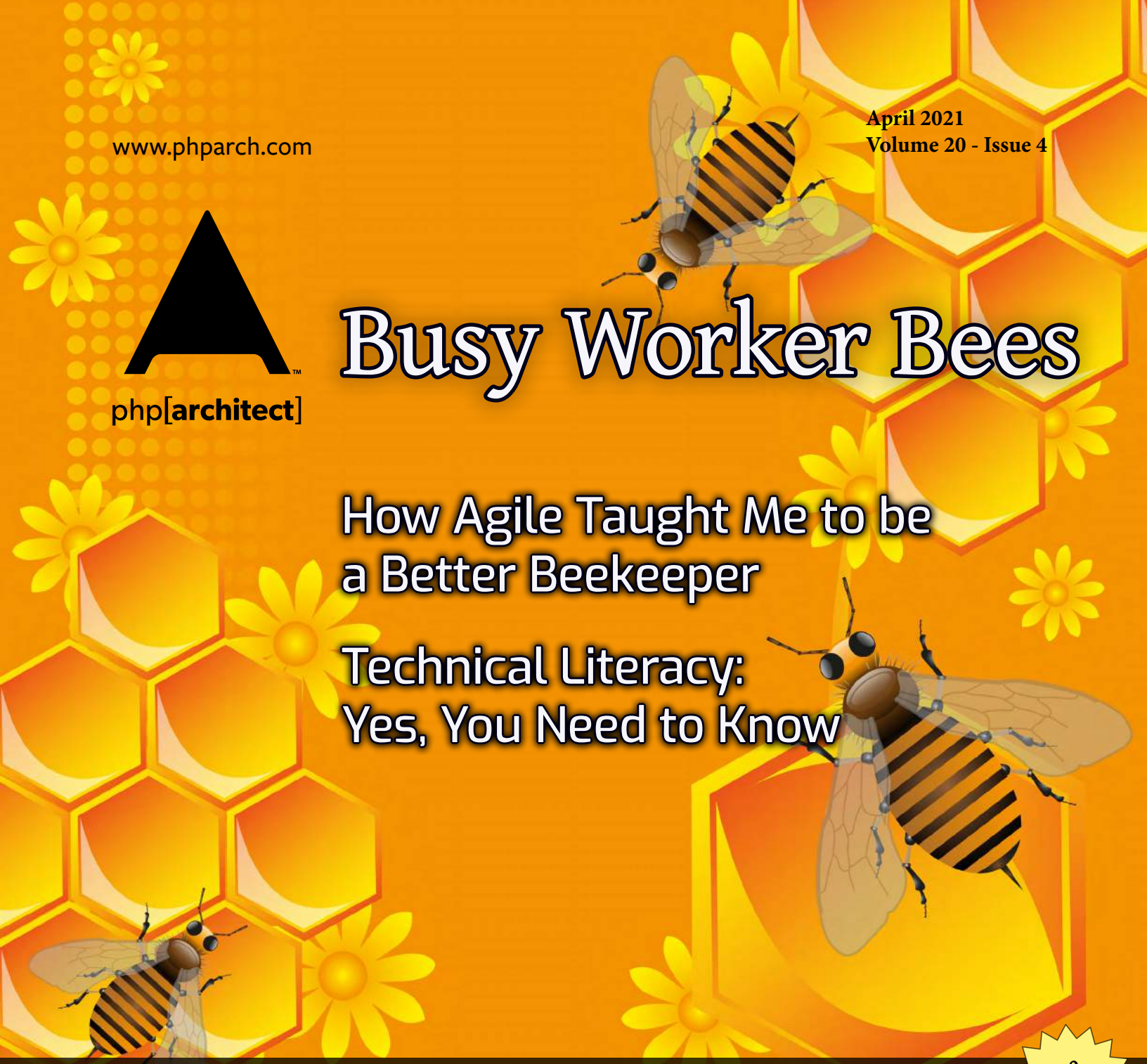
www.phparch.com

April 2021
Volume 20 - Issue 4



php[architect]

Busy Worker Bees

How Agile Taught Me to be a Better Beekeeper

Technical Literacy: Yes, You Need to Know

ALSO INSIDE

PHP Puzzles: Rock Paper Scissors

Education Station: Designing a REST API

Community Corner: A Bref of Fresh Air

The Workshop: Refactoring to an Object Store

Sustainable PHP: Machine Learning and Yoda

Security Corner: Basics of Password Hashing

finally: Doomsday Prepping

Free Sample Article

Basics of Password Hashing

Eric Mann

Every web application that allows users to authenticate needs to ensure their users' credentials are afforded the best protection possible. Conventionally, this is done by storing only the hash of a password rather than the password itself. Luckily, password hashing in PHP is secure, safe, and remarkably straightforward to implement.

Last month[1] we talked at length about different ways to make hashes of the identical origin string unique. When two users leverage the same authentication password, you don't want that to be evident in your data store. At the same time, it's helpful to back up a few steps and discuss password hashing and the algorithms that make it secure.

Absolutely Random

The most fundamental concept in cryptography that everyone needs to understand for hashing to make sense is randomness. In computing, there is no true sense of randomness. Computers are fully deterministic machines that are purpose-built to produce a specific output given a specific input. It's this predictable nature that makes computers reliable, but it's also what makes certain things hard.

The security behind cryptography is dependent on randomness. A "cryptographically secure" algorithm is one that renders its output completely indistinguishable from random noise. This is difficult to achieve in practice, but there are convenient extensions and methods available to PHP developers that take care of the problem for you.

Primarily, PHP developers should use the `password_hash()` function[2] to generate hashed passwords. Similarly, you should use the `password_verify()` function[3] to verify the hashes of any passwords you create. Both of these are securely implemented in PHP and rely on a cryptographically secure source of pseudo-randomness provided by the underlying operating system.

> *Many older tutorials reference the `mt_rand()` function as a pseudorandom number generator for use in PHP. It indeed produces pseudorandom numbers, but the function is not cryptographically secure. If you know the seed used to start the sequence, you can readily predict every number the function produces. This is a reminder that you should only rely on cryptographically secure functions for security operations in PHP.*

Time and Memory Factors

At the time of this writing, three algorithms are available for use with `password_hash()`. The default algorithm today is `CRYPT_BLOWFISH` and can be used by passing either `PASSWORD_DEFAULT` or `PASSWORD_BCRYPT` as a flag to the method:

```
$hash = password_hash($password, PASSWORD_DEFAULT);
```

Alternatively, if PHP was compiled with Argon2[4] support, you can pass `PASSWORD_ARGON2I` or `PASSWORD_ARGON2ID` to leverage the Argon2i or Argon2id algorithms, respectively. Each algorithm choice offers different advantages to you for security, but every algorithm will have an identifier coded into the hash output of the function. Even if the default algorithm in PHP changes in the future when you used `PASSWORD_DEFAULT` in the past, the system can detect the use of bcrypt and verify the hash properly. The `password_needs_rehash()` function can also indicate that you should upgrade and re-hash a password if you change the hashing algorithm in the future.

Developers can also tune each of these algorithms precisely to take as much time and use as many system resources as necessary to make hashing relatively slow on the server. The advantage of a slow hash is that, should your database ever be compromised and leaked, it will take an attacker a very long time to guess any particular hash by brute force.

> *A slow hash is not a problem for everyday user authentication. If a password takes one second to hash, then it will take one second for a user to authenticate with a correct password. Assuming an attacker were to try to guess a password by brute force, they would only be able to guess one password per second. Given six alphanumeric characters, there are 56,800,235,584 possible passwords. Again, guessing one password per second, it would take over 1,800 years to try all possible password combinations.*

The `PASSWORD_BCRYPT` algorithm supports a cost factor—it uses a cost of 10 by default. The higher the cost, the harder your machine will need to work to generate a hash. Given we want to target one hash per second, we can calculate the appropriate cost factor for a given server using the routine in Listing 1.

1 *Last month:* https://phparch.com/magazine/2021/march/

2 `password_hash()` *function:* https://php.net/password_hash

3 `password_verify()` *function:* https://php.net/password_verify

4 *Argon2:* https://en.wikipedia.org/wiki/Argon2

Listing 1.

```
$cost = 8;

do {
  $cost += 1;
  $start = microtime(true);
  password_hash('test', PASSWORD_BCRYPT,
                ['cost' => $cost]);
  $end = microtime(true);
} while (($end - $start) < 1);

echo sprintf('Time cost => %d', $cost);
```

Listing 2.

```
<?php
$cost = 2;

do {
  $cost += 1;
  $start = microtime(true);
  password_hash('test', PASSWORD_ARGON2I,
                ['time_cost' => $cost]);
  $end = microtime(true);
} while (($end - $start) < 1);

echo sprintf('Time cost => %d', $cost);
```

On my machine, the appropriate cost factor here is 15. This cost is much higher than the default of 10.

The Argon2 family of hashes uses similar settings but allows specifying both a time cost and a memory cost. For this example, we'll only tune the time cost, starting at a default of 2 (Listing 2).

For example, my machine suggests a time cost of 17, which is significantly higher than the hard-coded default. In order to control the speed of your hashing, your team must test the actual hashing speed of your server hardware and tune the process accordingly!

> *The Argon2id hash uses the same options and settings as Argon2i; you can use the same tuning script for both algorithms. On my machine, they both suggest a similar time factor.*

Secure Password Verification

When your users attempt to authenticate, we need to re-hash their supplied raw password and verify it hashes to the same output we've stored elsewhere in our database. While it might be tempting to do the hashing and string comparison directly, developers should never implement their own cryptographic operations.

Instead, we can leverage `password_verify()`, passing in both the user's supplied plaintext password and the stored password hash against which we want to compare it. Internally, this function will re-hash the plaintext password using the same algorithm and cost factor, then compare it securely to our known hash.

> *Secure string comparison is a separate topic we will discuss later. For now, know that merely comparing two strings with simple equality operators (===) is insufficient for cryptographically secure operations and trust that the core functions provided by PHP are making intelligent, secure decisions.*

```
$valid = password_verify($password, $stored_hash);
if (!$valid) {
  throw new UnauthorizedException('Invalid password!');
}
```

PHP will use information in the stored hash (i.e., algorithm and cost factors) and run the supplied plaintext through the same hashing operation it used before. If the output of the hash matches the known hash value, the function returns `true`. Otherwise, it returns `false`, and it's left to the developer to handle this negative return.

In Conclusion

Hashing itself is complex. We aren't detailing the algorithms used in this article as that discussion would be incredibly long and require a deep understanding of complicated mathematics. Luckily, PHP makes hashing passwords remarkably straightforward.

Do not spend time trying to write your own hashing algorithm. Do not try to invent your own password protection or verification protocols. Don't trust online tutorials that use any other hashing functions for passwords. Do leverage the built-in password hashing and verification functions provided by PHP. It's efficient and can be tuned to provide maximum protection to your end-users.

Eric is a seasoned web developer experienced with multiple languages and platforms. He's been working with PHP for more than a decade and focuses his time on helping developers get started and learn new skills with their tech of choice. You can reach out to him directly via Twitter: @EricMann

Related Reading

- *Security Corner: Cooking with Credentials* by Eric Mann, March 2021. https://phpa.me/security-mar-21
- *Security Corner: Credentials and Secrets Management* by Eric Mann, June 2019. https://phpa.me/security-corner-june-2019

Thank you for reading this issue!

Each issue is made possible by readers like you. If you're not already subscribed, start today to enjoy these benefits:

- DRM-free digital issues in PDF, EPUB, and Mobi formats
- Discord channel for subscribers.
- Access to all back issues starting in 2002.
- We also have print subscriptions available.

Subscribe or Renew Today
https://phparch.com/magazine