password_needs_rehash »
« password_get_info

- Manual de PHP
- Referencia de funciones
- Extensiones criptográficas
- Hash de contraseñas
- Funciones de hashing de contraseñas

Change language: Spanish

Submit a Pull Request Report a Bug

password_hash

(PHP 5 >= 5.5.0, PHP 7, PHP 8)

password_hash — Crea un hash de contraseña

Descripción ¶

password_hash(string `$password`, integer `$algo`, array `$options` = ?): string

password_hash() crea un nuevo hash de contraseña usando un algoritmo de hash fuerte de único sentido. **password_hash()** es compatible con crypt(). Por lo tanto, los hash de contraseñas creados con crypt() se pueden usar con **password_hash()**.

Actualmente se admiten los siguientes algoritmos:

- `PASSWORD_DEFAULT` - Usar el algoritmo bcrypt (predeterminado a partir de PHP 5.5.0). Observe que esta constante está diseñada para cambiar siempre que se añada un algoritmo nuevo y más fuerte a PHP. Por esta razón, la longitud del resultado de usar este identificador puede cambiar con el tiempo. Por lo tanto, se recomienda almacenar el resultado en una columna de una base de datos que pueda apliarse a más de 60 caracteres (255 caracteres sería una buena elección).
- `PASSWORD_BCRYPT` - Usar el algoritmo `CRYPT_BLOWFISH` para crear el hash. Producirá un hash estándar compatible con crypt() utilizando el identificador "$2y$". El resultado siempre será un string de 60 caracteres, o `false` en caso de error.

  Opciones admitidas:

  - `salt` - para proporcionar manualmente una sal a usar cuando se realiza el hash de la contraseña. Observe que esto sobrescribirá y prevendrá que una sal sea generada automáticamente.

    Si se omite, se generará una sal aleatoria mediante **password_hash()** para cada hash. Este es el modo de operación intencionado.

    Advertencia

    La opción salt está obsoleta a partir de PHP 7.0.0. Ahora se prefiere simplemente utilizar generada de manera predeterminada.

  - `cost` - denota el coste del algoritmo que debería usarse. Se pueden encontrar ejemplo de estos valores en la página de crypt().

    Si se omite, se usará el valor predeterminado `10`. Este es un buen coste de referencia, pero se podría considerar aumentarlo dependiendo del hardware.

Parámetros ¶

`password`

La contraseña del usuario.

Precaución

El uso de **`PASSWORD_BCRYPT`** como el algoritmo resultará en el truncamiento del parámetro `password` a un máximo de 72 caracteres de longitud.

`algo`

A constante del algoritmo de contraseñas indicando qué algoritmo utilizar para crear el hash de la contraseña.

`options`

Un array asociativo de opciones. Véanse las constantes de algoritmos de contraseñas para la documentación sobre las opociones admitidas de cada algoritmo.

Si no se indica, se creará una sal aleatoria y el coste algorítmico por defecto será utilizado.

Valores devueltos ¶

Devuelve el hash de la contraseña, o **`false`** en caso de error.

El algoritmo, coste y sal usados son devueltos como parte del hash. Por lo tanto, toda la información que es necesaria para verificar el hash, está incluida en él. Esto permite que la función password_verify() verifique el hash sin tener que almacenar por separado la información de la sal o del algoritmo.

Ejemplos ¶

Ejemplo #1 Ejemplo de password_hash()

```
<?php
/**
 * Queremos crear un hash de nuestra contraseña uando el algoritmo DEFAULT actual.
 * Actualmente es BCRYPT, y producirá un resultado de 60 caracteres.
 *
 * Hay que tener en cuenta que DEFAULT puede cambiar con el tiempo, por lo que debería prepararse
 * para permitir que el almacenamento se amplíe a más de 60 caracteres (255 estaría bien)
 */
echo password_hash("rasmuslerdorf", PASSWORD_DEFAULT)."\n";
?>
```

El resultado del ejemplo sería algo similar a:

```
$2y$10$.vGA1O9wmRjrwAVXD98HNOgsNpDczlqm3Jq7KnEd1rVAGv3Fykk1a
```

Ejemplo #2 Ejemplo de password_hash() estableciendo el coste manualmente

```
<?php
/**
 * En este caso, queremos aumentar el coste predeterminado de BCRYPT a 12.
 * Observe que también cambiamos a BCRYPT, que tendrá siempre 60 caracteres.
 */
$opciones = [
    'cost' => 12,
];
```

```
echo password_hash("rasmuslerdorf", PASSWORD_BCRYPT, $opciones)."\n";
?>
```

El resultado del ejemplo sería algo similar a:

```
$2y$12$QjSH496pcT5CEbzjD/vtVeH03tfHKFy36d4J0Ltp3lRtee9HDxY3K
```

Ejemplo #3 Ejemplo de password_hash() estableciendo la sal manualmente

```
<?php
/**
 * Observe que la sal se genera aleatoriamente aquí.
 * No use nunca una sal estática o una que no se genere aleatoriamente.
 *
 * Para la GRAN mayoría de los casos de uso, dejar que password_hash genere la sal aleatoriamente
 */
$opciones = [
    'cost' => 11,
    'salt' => mcrypt_create_iv(22, MCRYPT_DEV_URANDOM),
];
echo password_hash("rasmuslerdorf", PASSWORD_BCRYPT, $opciones)."\n";
?>
```

El resultado del ejemplo sería algo similar a:

```
$2y$11$q5MkhSBtlsJcNEVsYh64a.aCluzHnGog7TQAKVmQwO9C8xb.t89F.
```

Ejemplo #4 Ejemplo de password_hash() buscando un buen coste

```
<?php
/**
 * Este código evaluará el servidor para determinar el coste permitido.
 * Se establecerá el mayor coste posible sin disminuir demasiando la velocidad
 * del servidor. 8-10 es una buena referencia, y más es bueno si los servidores
 * son suficientemente rápidos. El código que sigue tiene como objetivo un tramo de
 * ≤ 50 milisegundos, que es una buena referencia para sistemas con registros interactivos.
 */
$timeTarget = 0.05; // 50 milisegundos

$coste = 8;
do {
    $coste++;
    $inicio = microtime(true);
    password_hash("test", PASSWORD_BCRYPT, ["cost" => $coste]);
    $fin = microtime(true);
} while (($fin - $inicio) < $timeTarget);

echo "Coste apropiado encontrado: " . $coste . "\n";
?>
```

El resultado del ejemplo sería algo similar a:

```
Coste apropiado encontrado: 10
```

Notas ¶

Precaución

Se recomienda encarecidamente que no se genere una sal propia para esta función. Creará una sal segura automáticamente si no se especifica una.

Como se ha observado arriba, proporcionar la opción `salt` en PHP 7.0 generará una advertencia de obsolescencia. El soporte para proporcionar una sal manualmente podría ser eliminado en una versión futura de future.

> **Nota**:
>
> Se recomienda probar esta función en los servidores que se estén usando, y ajustar el parámetro de coste para que la ejecución de la función tome menos de 100 milisegundos en sistemas interactivos. El script en el ejemplo de antes le ayudará a elegir un buen valor de coste para su hardware.
>
> **Nota**: Las actualizaciones para los algoritmos admitidos por esta función (o los cambios al predeterminado) deben serguir las siguientes reglas:
>
> - Cualquier algoritmo nuevo debe estar en el núcleo por lo menos una versión completa de PHP antes de convertirse en predeterminado. Por lo que si, por ejemplo, un nuevo algoritmo se añade en la 7.5.5, no sería apto para ser el predeterminado hasta la 7.7 (ya que la 7.6 sería la primera versión completa). Pero si se añadió un algoritmo diferente en la 7.6.0, también sería apto para ser el predeterminado en la 7.7.0.
> - El predeterminado debería cambiar únicamente en una versión completa (7.3.0, 8.0.0, etc.) y no en una versión de revisión. La única excepción es la emergencia de que se encuentre un fallo de seguridad crítico en el predeterminado actual.

Ver también ¶

- password_verify() - Comprueba que la contraseña coincida con un hash
- crypt() - Hash de cadenas de un sólo sentido
- » implementación en el espacio de usuario

+ add a note

User Contributed Notes 12 notes

up
down
169
martinstoeckli ¶
9 years ago

```
There is a compatibility pack available for PHP versions 5.3.7 and later, so you don't have to
wait on version 5.5 for using this function. It comes in form of a single php file:
https://github.com/ircmaxell/password_compat
```

up
down
116
phpnetcomment201908 at lucb1e dot com ¶
3 years ago

```
Since 2017, NIST recommends using a secret input when hashing memorized secrets such as passwords.
By mixing in a secret input (commonly called a "pepper"), one prevents an attacker from brute-
forcing the password hashes altogether, even if they have the hash and salt. For example, an SQL
injection typically affects only the database, not files on disk, so a pepper stored in a config
file would still be out of reach for the attacker. A pepper must be randomly generated once and
can be the same for all users. Many password leaks could have been made completely useless if site
owners had done this.

Since there is no pepper parameter for password_hash (even though Argon2 has a "secret" parameter,
```

PHP does not allow to set it), the correct way to mix in a pepper is to use hash_hmac(). The "add note" rules of php.net say I can't link external sites, so I can't back any of this up with a link to NIST, Wikipedia, posts from the security stackexchange site that explain the reasoning, or anything... You'll have to verify this manually. The code:

```
// config.conf
pepper=c1isvFdxMDdmjOlvxpecFw

<?php
// register.php
$pepper = getConfigVariable("pepper");
$pwd = $_POST['password'];
$pwd_peppered = hash_hmac("sha256", $pwd, $pepper);
$pwd_hashed = password_hash($pwd_peppered, PASSWORD_ARGON2ID);
add_user_to_database($username, $pwd_hashed);
?>

<?php
// login.php
$pepper = getConfigVariable("pepper");
$pwd = $_POST['password'];
$pwd_peppered = hash_hmac("sha256", $pwd, $pepper);
$pwd_hashed = get_pwd_from_db($username);
if (password_verify($pwd_peppered, $pwd_hashed)) {
    echo "Password matches.";
}
else {
    echo "Password incorrect.";
}
?>
```

Note that this code contains a timing attack that leaks whether the username exists. But my note was over the length limit so I had to cut this paragraph out.

Also note that the pepper is useless if leaked or if it can be cracked. Consider how it might be exposed, for example different methods of passing it to a docker container. Against cracking, use a long randomly generated value (like in the example above), and change the pepper when you do a new install with a clean user database. Changing the pepper for an existing database is the same as changing other hashing parameters: you can either wrap the old value in a new one and layer the hashing (more complex), you compute the new password hash whenever someone logs in (leaving old users at risk, so this might be okay depending on what the reason is that you're upgrading).

Why does this work? Because an attacker does the following after stealing the database:

```
password_verify("a", $stolen_hash)
password_verify("b", $stolen_hash)
...
password_verify("z", $stolen_hash)
password_verify("aa", $stolen_hash)
etc.
```

(More realistically, they use a cracking dictionary, but in principle, the way to crack a password hash is by guessing. That's why we use special algorithms: they are slower, so each verify() operation will be slower, so they can try much fewer passwords per hour of cracking.)

Now what if you used that pepper? Now they need to do this:

```
password_verify(hmac_sha256("a", $secret), $stolen_hash)
```

Without that $secret (the pepper), they can't do this computation. They would have to do:

password_verify(hmac_sha256("a", "a"), $stolen_hash)
password_verify(hmac_sha256("a", "b"), $stolen_hash)
...
etc., until they found the correct pepper.

If your pepper contains 128 bits of entropy, and so long as hmac-sha256 remains secure (even MD5 is technically secure for use in hmac: only its collision resistance is broken, but of course nobody would use MD5 because more and more flaws are found), this would take more energy than the sun outputs. In other words, it's currently impossible to crack a pepper that strong, even given a known password and salt.

up
down
44

nicoSWD ¶
9 years ago

I agree with martinstoeckli,

don't create your own salts unless you really know what you're doing.

By default, it'll use /dev/urandom to create the salt, which is based on noise from device drivers.

And on Windows, it uses CryptGenRandom().

Both have been around for many years, and are considered secure for cryptography (the former probably more than the latter, though).

Don't try to outsmart these defaults by creating something less secure. Anything that is based on rand(), mt_rand(), uniqid(), or variations of these is *not* good.

up
down
18

Lyo Mi ¶
6 years ago

Please note that password_hash will ***truncate*** the password at the first NULL-byte.

http://blog.ircmaxell.com/2015/03/security-issue-combining-bcrypt-with.html

If you use anything as an input that can generate NULL bytes (sha1 with raw as true, or if NULL bytes can naturally end up in people's passwords), you may make your application much less secure than what you might be expecting.

The password
$a = "\01234567";
is zero bytes long (an empty password) for bcrypt.

The workaround, of course, is to make sure you don't ever pass NULL-bytes to password_hash.

up
down
27

Cloxy ¶
9 years ago

You can produce the same hash in php 5.3.7+ with crypt() function:

<?php

```
$salt = mcrypt_create_iv(22, MCRYPT_DEV_URANDOM);
$salt = base64_encode($salt);
$salt = str_replace('+', '.', $salt);
$hash = crypt('rasmuslerdorf', '$2y$10$'.$salt.'$');

echo $hash;

?>
```

up
down
15

martinstoeckli ¶

9 years ago

In most cases it is best to omit the salt parameter. Without this parameter, the function will generate a cryptographically safe salt, from the random source of the operating system.

up
down
11

Mike Robinson ¶

8 years ago

For passwords, you generally want the hash calculation time to be between 250 and 500 ms (maybe more for administrator accounts). Since calculation time is dependent on the capabilities of the server, using the same cost parameter on two different servers may result in vastly different execution times. Here's a quick little function that will help you determine what cost parameter you should be using for your server to make sure you are within this range (note, I am providing a salt to eliminate any latency caused by creating a pseudorandom salt, but this should not be done when hashing passwords):

```
<?php
/**
* @Param int $min_ms Minimum amount of time in milliseconds that it should take
* to calculate the hashes
*/
function getOptimalBcryptCostParameter($min_ms = 250) {
    for ($i = 4; $i < 31; $i++) {
        $options = [ 'cost' => $i, 'salt' => 'usesomesillystringforsalt' ];
        $time_start = microtime(true);
        password_hash("rasmuslerdorf", PASSWORD_BCRYPT, $options);
        $time_end = microtime(true);
        if (($time_end - $time_start) * 1000 > $min_ms) {
            return $i;
        }
    }
}
echo getOptimalBcryptCostParameter(); // prints 12 in my case
?>
```

up
down
0

ms1 at rdrecs dot com ¶

3 years ago

Timing attacks simply put, are attacks that can calculate what characters of the password are due to speed of the execution.

More at...
https://paragonie.com/blog/2015/11/preventing-timing-attacks-on-string-comparison-with-double-hmac-strategy

I have added code to phpnetcomment201908 at lucb1e dot com's suggestion to make this possible "timing attack" more difficult using the code phpnetcomment201908 at lucb1e dot com posted.

```
$pph_strt = microtime(true);

//...
/*The code he posted for login.php*/
//...

$end = (microtime(true) - $pph_strt);

$wait = bcmul((1 - $end), 1000000);  // usleep(250000) 1/4 of a second

usleep ( $wait );

echo "<br>Execution time:".(microtime(true) - $pph_strt)."; ";
```

Note I suggest changing the wait time to suit your needs but make sure that it is more than than the highest execution time the script takes on your server.

Also, this is my workaround to obfuscate the execution time to nullify timing attacks. You can find an in-depth discussion and more from people far more equipped than I for cryptography at the link I posted. I do not believe this was there but there are others. It is where I found out what timing attacks were as I am new to this but would like solid security.

up
down
-5

php dot net at marksim dot org ¶
2 years ago

regarding the sentence "...database column that can expand beyond 60 characters (255 characters would be a good choice). "

Considering future hash length increase by factor *2 and considering databases to start counting with 1, a password length of 256 characters (not 255) would probably be the better choice :)

up
down
-9

Anonymous ¶
3 years ago

According to the draft specification, Argon2di is the recommended mode of operation:

> 9.4. Recommendations
>
>   The Argon2id variant with t=1 and maximum available memory is
>   recommended as a default setting for all environments. This setting
>   is secure against side-channel attacks and maximizes adversarial
>   costs on dedicated bruteforce hardware.

source: https://tools.ietf.org/html/draft-irtf-cfrg-argon2-06#section-9.4

up
down
-8

Anonymous ¶
2 years ago

To use argon, follow these steps:

```
```

```
git clone https://github.com/p-h-c/phc-winner-argon2
cd phc-winner-argon2 && make && make install
apt install libsodium-dev
cd ~/php-7.4.5 // Your php installation source code
./configure [YOUR_EXISTING_CONFIGURE_COMMANDS] --with-password-argon2 --with-sodium
```

up
down
-19
hman ¶
3 years ago

I believe a note should be added about the compatibility of crypt() and password_hash().

My tests showed that yes, password_verify can also take hashes generated by crypt - as well as those from password_hash. But vice versa this is not true...

You cannot put hashes generated by password_hash into crypt for comparing them themselves, when used as the salt for crypt, as was recommended years ago (compare user entry with user crypt(userentry,userentry). No big deal, but it means that password checking routines MUST immediately be rewritten to use password_hash...

You cannot start using password_hash for hash generation without also altering the password check routine!

So the word "compatible" should be, IMHO, ammended with a word of caution, hinting the reader, that compatibility here is a one-way street.

+ add a note

- Funciones de hashing de contraseñas
  - password_algos
  - password_get_info
  - password_hash
  - password_needs_rehash
  - password_verify