de dos caracteres (DES), o una de doce caracteres (MD5), dependiendo de la disponibilidad de la función crypt() de MD5. PHP establece una constante llamada **CRYPT_SALT_LENGTH** la cual indica la sal de mayor longitud permitida por los hash disponibles.

La función **crypt()** estándar basada en DES devuelve la sal como los primeros dos caracteres de la salida. También utiliza solamente los primeros ocho caracteres de `str`, por lo que cadenas más largas que empiecen con los mismos ocho caracteres, generarán el mismo resultado (cuando se utiliza la misma sal).

En sistemas donde la función crypt() soporta múltiples tipos de hash, las siguientes contantes se establecen en 0 o 1, dependiendo de que si el tipo dado está disponible:

- **CRYPT_STD_DES** - Hash estándar basado en DES con un salt de dos caracteres del alfabeto "./0-9A-Za-z". Utilizar caracteres no válidos en el salt causará que crypt() falle.
- **CRYPT_EXT_DES** - Hash extendido basado en DES. El "salt" es un string de 9 caracteres que consiste en un guión bajo seguido de 4 bytes del conteo de iteraciones y 4 bytes del salt. Estos están codificados como caracteres imprimibles, 6 bits por caracter, por lo menos, el carácter significativo primero. Los valores del 0 al 63 son codificados como "./0-9A-Za-z". Utilizar caracteres no válidos en el salt causará que crypt() falle.
- **CRYPT_MD5** - Hash MD5 con un salt de doce caracteres comenzando con 1
- **CRYPT_BLOWFISH** - Hash con Blowfish con un salt como sigue: "$2a$", "$2x$" o "$2y$", un parámetro de coste de dos dígitos, "$", y 22 caracteres del alfabeto "./0-9A-Za-z". Utilizar caracteres fuera de este rango en el salt causará que crypt() devuelva una cadena de longitud cero. El parámetro de coste de dos dígitos es el logaritmo en base 2 de la cuenta de la iteración del algoritmo hach basado en Blowfish subyacente, y debe estar en el rango 04-31; los valores fuera de este rango causarán que crypt() falle. Los hashes "$2x$" potencialmente débiles; los hashes "$2a$" son compatible y y mitigan esta debilidad. Para los nuevos hashes, se debe utilizar "$2y$".
- **CRYPT_SHA256** - Hash SHA-256 con un salt de dieciséis caracteres prefijado con 5. Si el strig del salt inicia con 'rounds=<N>$', el valor numérico de N se utiliza para indicar cuantas veces el bucle del hash se debe ejecutar, muy similar al parámetro de costo en Blowfish. El número de rondas por defecto es 5000, hay un mínimo de 1000 y un máximo de 999,999,999. Cualquier selección de N por fuera de este rango será truncada al límite más cercano.
- **CRYPT_SHA512** - Hash SHA-512 con un salt de dieciséis caracteres prefijado con 6. Si el strig del salt inicia con 'rounds=<N>$', el valor numérico de N se utiliza para indicar cuantas veces el bucle del hash se debe ejecutar, muy similar al parámetro de costo en Blowfish. El número de rondas por defecto es 5000, hay un mínimo de 1000 y un máximo de 999,999,999. Cualquier selección de N por fuera de este rango será truncada al límite más cercano.

  Nota:

  A partir de PHP 5.3.0, PHP contiene su propia implementación y la utilizará si el sistema carece de soporte para uno o varios de los algoritmos.

Parámetros ¶

`str`

El string al que realizarle el hash.

Precaución

El uso del algoritmo **CRYPT_BLOWFISH** resultará en que el parámetro `str` sea truncado a una longitud máxima de 72 caracteres.

`salt`

Un string opcional de salt para la base del hash. Si no se proporciona, el comportamiento se define por la aplicación del algoritmo y puede conducir a resultados inesperados.

Valores devueltos ¶

Devuelve un string con el hash o un string que es más corto que 13 caracteres y que se garantiza que difiere del salt en caso de fallo.

Advertencia

Cuando se validan contraseñas, se debería usar una función de comparación de strings que no sea vulnerable a ataques de temporización para comparar la salida de **crypt()** con el hash conocido anteriormente. PHP 5.6 y siguientes proporcionan hash_equals() para este propósito.

Historial de cambios ¶

Versión	Descripción
5.6.5	Cuando se da la cadena de fallo "*0" como `salt`, ahora se devuelve "*1" por consistencia con otras implementaciones de crypt. Antes de esta versión, PHP 5.6 devolvía incorrectamente un hash DES.
5.6.0	Se emite una advertencia de seguridad E_NOTICE si se omite `salt`.
5.5.21	Cuando se da la cadena de fallo "*0" como `salt`, ahora se devuelve "*1" por consistencia con otras implementaciones de crypt. Antes de esta versión, PHP 5.5 (y ramificaciones anteriores) devolvía incorrectamente un hash DES.
5.3.7	Se añadieron los modos de Blowfish `$2x$` y `$2y$` para tratar con ataques de bit alto potenciales.
5.3.2	Agregado el crypt SHA-256 y SHA-512 basado en la » implementación de Ulrich Drepper.
5.3.2	Corregido el comportamiento de Blowfish sobre rondas no válidas para devolver el string "failure" ("*0" or "*1"), en lugar de caer al DES.
5.3.0	PHP ahora contiene su propia implementación de los algoritmos MD5, DES estándar, DES extendido y Blowfish y los utilizará si el sistema carece de soporte para uno o varios de los algoritmos.

Ejemplos ¶

Ejemplo #1 crypt() examples

```
<?php
$hashed_password = crypt('mypassword'); // dejar que el salt se genera automáticamente

/* Se deben pasar todos los resultados de crypt() como el salt para la comparación de una
   contraseña, para evitar problemas cuando diferentes algoritmos hash son utilizados. (Como
   se dice arriba, el hash estándar basado en DES utiliza un salt de 2
   caracteres, pero el hash basado en MD5 utiliza 12.) */
if (hash_equals($hashed_password, crypt($user_input, $hashed_password))) {
   echo "¡Contraseña verificada!";
}
?>
```

Ejemplo #2 Using crypt() with htpasswd

```
<?php
// Establece la contraseña
$password = 'mypassword';

// Obtiene el hash, dejando que el salt sea generado be automáticamente
$hash = crypt($password);
?>
```

Ejemplo #3 Using crypt() with different hash types

```
<?php
/* Estas sales son solamente ejemplos, y no deberían usarse textualmente en su código.
   Debería generar una sal distinta correctamente formada para cada contraseña.
*/
if (CRYPT_STD_DES == 1) {
    echo 'Standard DES: ' . crypt('rasmuslerdorf', 'rl') . "\n";
}

if (CRYPT_EXT_DES == 1) {
    echo 'Extended DES: ' . crypt('rasmuslerdorf', '_J9..rasm') . "\n";
}

if (CRYPT_MD5 == 1) {
    echo 'MD5:          ' . crypt('rasmuslerdorf', '$1$rasmusle$') . "\n";
}

if (CRYPT_BLOWFISH == 1) {
    echo 'Blowfish:     ' . crypt('rasmuslerdorf', '$2a$07$usesomesillystringforsalt$') . "\n";
}

if (CRYPT_SHA256 == 1) {
    echo 'SHA-256:      ' . crypt('rasmuslerdorf', '$5$rounds=5000$usesomesillystringforsalt$') . "\n";
}

if (CRYPT_SHA512 == 1) {
    echo 'SHA-512:      ' . crypt('rasmuslerdorf', '$6$rounds=5000$usesomesillystringforsalt$') . "\n";
}
?>
```

El resultado del ejemplo sería algo similar a:

```
Standard DES: rl.3StKT.4T8M
Extended DES: _J9..rasmBYk8r9AiWNc
MD5:          $1$rasmusle$rISCgZzpwk3UhDidwXvin0
Blowfish:     $2a$07$usesomesillystringfore2uDLvp1Ii2e./U9C8sBjqp8I90dH6hi
SHA-256:      $5$rounds=5000$usesomesillystri$KqJWpanXZHKq2BOB43TSaYhEWsQ1Lr5QNyPCDH/Tp.6
SHA-512:      $6$rounds=5000$usesomesillystri$D4IrlXatmP7rx3P3InaxBeoomnAihCKRVQP22JZ6EY47Wc6BkroIuUUBOov1i.S5KPgErtP/EN5mcO.ChWQW21
```

Notas ¶

> **Nota**: No hay función de descifrado, ya que **crypt()** utiliza un algoritmo de un solo sentido.

Ver también ¶

- hash_equals() - Comparación de strings segura contra ataques de temporización
- password_hash() - Crea un hash de contraseña
- md5() - Calcula el 'hash' md5 de un string
- La extensión Mcrypt
- La página man de la función crypt de Unix para más información

+ add a note

User Contributed Notes 8 notes

up
down
69
bob dot orr at mailinator dot com ¶
7 years ago
The #2 comment on this comments page (as of Feb 2015) is 9 years old and recommends phpass. I have independently security audited this product and, while it continues to be recommended for password security, it is actually insecure and should NOT be used. It hasn't seen any updates in years (still at v0.3) and there are more recent alternatives such as using the newer built-in PHP password_hash() function that are much better. Everyone, please take a few moments to confirm what I'm saying is accurate (i.e. review the phpass code for yourself) and then click the down arrow to sink the phpass comment to the bottom. You'll be increasing security across the Internet by doing so.

For those who want details: md5() with microtime() are a fallback position within the source code of phpass. Instead of terminating, it continues to execute code. The author's intentions of trying to work everywhere are admirable but, when it comes to application security, that stance actually backfires. The only correct answer in a security context is to terminate the application rather than fallback to a weak position that can potentially be exploited (usually by forcing that weaker position to happen).

up
down
24
Marten Jacobs ¶
8 years ago
As I understand it, blowfish is generally seen a secure hashing algorithm, even for enterprise use (correct me if I'm wrong). Because of this, I created functions to create and check secure password hashes using this algorithm, and using the (also deemed cryptographically secure) openssl_random_pseudo_bytes function to generate the salt.

```
<?php
/*
* Generate a secure hash for a given password. The cost is passed
* to the blowfish algorithm. Check the PHP manual page for crypt to
* find more information about this setting.
*/
function generate_hash($password, $cost=11){
        /* To generate the salt, first generate enough random bytes. Because
         * base64 returns one character for each 6 bits, the we should generate
         * at least 22*6/8=16.5 bytes, so we generate 17. Then we get the first
         * 22 base64 characters
         */
        $salt=substr(base64_encode(openssl_random_pseudo_bytes(17)),0,22);
        /* As blowfish takes a salt with the alphabet ./A-Za-z0-9 we have to
         * replace any '+' in the base64 string with '.'. We don't have to do
         * anything about the '=', as this only occurs when the b64 string is
         * padded, which is always after the first 22 characters.
         */
        $salt=str_replace("+",".",$salt);
        /* Next, create a string that will be passed to crypt, containing all
         * of the settings, separated by dollar signs
         */
        $param='$'.implode('$',array(
                "2y", //select the most secure version of blowfish (>=PHP 5.3.7)
                str_pad($cost,2,"0",STR_PAD_LEFT), //add the cost in two digits
                $salt //add the salt
        ));

        //now do the actual hashing
        return crypt($password,$param);
}

/*
* Check the password against a hash generated by the generate_hash
* function.
*/
function validate_pw($password, $hash){
        /* Regenerating the with an available hash as the options parameter should
         * produce the same hash if the same password is passed.
         */
        return crypt($password, $hash)==$hash;
}
?>
```

up
down
9

steve at tobtu dot com ¶
9 years ago
To generate salt use mcrypt_create_iv() not mt_rand() because no matter how many times you call mt_rand() it will only have at most 32 bits of entropy. Which you will start seeing salt collisions after about 2^16 users. mt_rand() is seeded poorly so it should happen sooner.

For bcrypt this will actually generate a 128 bit salt:
```
<?php $salt = strtr(base64_encode(mcrypt_create_iv(16, MCRYPT_DEV_URANDOM)), '+', '.'); ?>
```

*** Bike shed ***
The last character in the 22 character salt is 2 bits.
base64_encode() will have these four character "AQgw"
bcrypt will have these four character ".Oeu"

You don't need to do a full translate because they "round" to different characters:
```
echo crypt('', '$2y$05$.....................A') . "\n";
echo crypt('', '$2y$05$.....................Q') . "\n";
echo crypt('', '$2y$05$.....................g') . "\n";
echo crypt('', '$2y$05$.....................w') . "\n";
```

```
$2y$05$.....................J2ihDv8vVf7QZ9BsaRrKyqs2tkn55Yq
$2y$05$.....................O/jw2XygQa2.LrIT7CFCBQowLowDP6Y.
$2y$05$.....................eDOx4wMcy7WU.kE21W6nJfdMimsBE3V6
$2y$05$.....................uMMcgjnOELIa6oydRivPkiMrBG8.aFp.
```
up
down
5
kaminski at istori dot com ¶
11 years ago
Here is an expression to generate pseudorandom salt for the CRYPT_BLOWFISH hash type:

```
<?php $salt = substr(str_replace('+', '.', base64_encode(pack('N4', mt_rand(), mt_rand(), mt_rand(), mt_rand()))), 0, 22); ?>
```

It is intended for use on systems where mt_getrandmax() == 2147483647.

The salt created will be 128 bits in length, padded to 132 bits and then expressed in 22 base64 characters. (CRYPT_BLOWFISH only uses 128 bits for the salt, even though there are 132 bits in 22 base64 characters. If you examine the CRYPT_BLOWFISH input and output, you can see that it ignores the last four bits on input, and sets them to zero on output.)

Note that the high-order bits of the four 32-bit dwords returned by mt_rand() will always be zero (since mt_getrandmax == 2^31), so only 124 of the 128 bits will be pseudorandom. I found that acceptable for my application.
up
down
-2
jette at nerdgirl dot dk ¶
9 years ago
The crypt() function cant handle plus signs correctly. So if for example you are using crypt in a login function, use urlencode on the password first to make sure that the login procedure can handle any character:

```
<?php
$user_input =  '12+#æ345';
$pass = urlencode($user_input));
$pass_crypt = crypt($pass);

if ($pass_crypt == crypt($pass, $pass_crypt)) {
  echo "Success! Valid password";
} else {
  echo "Invalid password";
}
?>
```
up
down
-13
Joey ¶
5 years ago
While the documentation says that crypt will fail for DES if the salt is invalid, this turns out to not be the case.

The crypt function will accept any string of two characters or more for DES as long as it doesn't match the pattern for any other hashing schema. The remaining characters will be ignored.
up
down
-18
Anonymous ¶
5 years ago
steve at tobtu dot com was right 4 years ago, but now mcrypt_create_iv() (and bcrypt in general) is deprecated!

Use random_bytes() instead:

```
<?php
$salt = base64_encode(random_bytes(16));
```

up
down
-23
ian+php dot net at eiloart dot ocm ¶
8 years ago

If you're stuck with CRYPT_EXT_DES, then you'll want to pick a number of iterations: the 2nd-5th characters of the "salt".

My experimentation suggests that the 5th character is the most significant. A '.' is a zero and 'Z' is the highest value. Using all dots will create an error: all passwords will be encrypted to the same value.

Here are some encryption timings (in seconds) that I obtained, with five different iteration counts over the same salt, and the same password, on a quad core 2.66GHz Intel Xeon machine.

```
_1111 time: 0.15666794776917
_J9.Z time: 1.8860530853271
_J9.. time: 0.00015401840209961
_...Z time: 1.9095730781555
_ZZZZ time: 1.9124970436096
_...A time: 0.61211705207825
```

I think a half a second is reasonable for an application, but for the back end authentication? I'm not so sure: there's a significant risk of overloading the back end if we're getting lots of authentication requests.

+ add a note

- Funciones de strings
  - addcslashes
  - addslashes
  - bin2hex
  - chop
  - chr
  - chunk_split
  - convert_uudecode
  - convert_uuencode
  - count_chars
  - crc32
  - crypt
  - echo
  - explode
  - fprintf
  - get_html_translation_table
  - hebrev
  - hex2bin
  - html_entity_decode
  - htmlentities
  - htmlspecialchars_decode
  - htmlspecialchars
  - implode
  - join
  - lcfirst
  - levenshtein
  - localeconv
  - ltrim
  - md5_file
  - md5
  - metaphone
  - money_format
  - nl_langinfo
  - nl2br
  - number_format
  - ord
  - parse_str
  - print
  - printf
  - quoted_printable_decode
  - quoted_printable_encode
  - quotemeta
  - rtrim
  - setlocale
  - sha1_file
  - sha1
  - similar_text
  - soundex
  - sprintf
  - sscanf

- str_contains
- str_ends_with
- str_getcsv
- str_ireplace
- str_pad
- str_repeat
- str_replace
- str_rot13
- str_shuffle
- str_split
- str_starts_with
- str_word_count
- strcasecmp
- strchr
- strcmp
- strcoll
- strcspn
- strip_tags
- stripcslashes
- stripos
- stripslashes
- stristr
- strlen
- strnatcasecmp
- strnatcmp
- strncasecmp
- strncmp
- strpbrk
- strpos
- strrchr
- strrev
- strripos
- strrpos
- strspn
- strstr
- strtok
- strtolower
- strtoupper
- strtr
- substr_compare
- substr_count
- substr_replace
- substr
- trim
- ucfirst
- ucwords
- utf8_decode
- utf8_encode
- vfprintf
- vprintf
- vsprintf
- wordwrap

- Deprecated
  - convert_cyr_string
  - hebrevc