Encriptación y contraseñas en PHP

La encriptación de contraseñas para las cuentas de los usuarios de una aplicación web es una de las medidas más importantes para la seguridad

Contenido modificable

Si ves errores o quieres modificar/añadir contenidos, puedes crear un pull request. Gracias

Desde el principio **PHP** ha sido un **lenguaje de programación para la construcción de sitios web**. Esa idea permanece en el núcleo del lenguaje, y por eso es tan popular para la **construcción de aplicaciones web**. Cuando se creó en los años 90, el término **aplicación web** no existía aún, por lo que la **protección de contraseñas para cuentas de usuarios** no era algo en lo que estuviera centrado.

Han pasado muchos años desde entonces y actualmente es impensable una aplicación web que no proteja las cuentas de los usuarios con **contraseñas**. Es fundamental para cualquier programador hacer que estas contraseñas tengan una **encriptación segura y eficiente**. **PHP 5.5** añadió una nueva librería llamada Hash de contraseñas para la **encriptación de contraseñas**, con funciones que facilitan la tarea y utilizan los últimos métodos más eficaces.

Indice de contenido

1. La importancia de los hashes seguros

Siempre hay que guardar las contraseñas encriptadas mediante un **algoritmo de encriptación** como el **algoritmo hashing** para hacer imposible a alguien que acceda a una base de datos conseguir averiguar la contraseña. Esto no es sólo para proteger a los usuarios frente a algún atacante sino también frente a los propios empleados de la aplicación.

Mucha gente utiliza las mismas contraseñas para muchas **aplicaciones web**. Si alguien accede a la dirección de email y contraseña de un usuario, probablemente pueda hacerlo en muchas otras aplicaciones.

Los hashes no se crean iguales, se emplean algoritmos muy distintos para crear un hash. Los dos más usados en el pasado son **MD5** y **SHA-1**. Los ordenadores de hoy en día pueden **crackear** fácilmente estos **algoritmos**. Dependiendo de la **complejidad y longitud de la contraseña**, se puede crackear en menos de una hora con los dos algoritmos nombrados (los ratios son 3650 millones de cálculos por segundo con MD5 y 1360 millones por segundo con SHA-1).

Por eso es importante usar **algoritmos complejos**. Si el hash es más largo reduce el **riesgo de colisiones entre contraseñas** (dos frases generando el mismo hash), pero también conviene que la aplicación se tome el **tiempo necesario para generar el hash**. Esto es porque el usuario apenas notará un segundo o dos más de tiempo de carga al logearse, pero se consigue que crackearlo tome muchisimo más tiempo, en case de que sea posible.

También es necesario protegerse frente a las **Rainbow Tables**. Las Rainbow Tables, como a la **MD5** que puede verse en este enlace, son **tablas de búsqueda inversa para hashes**. El creador de las tablas precalcula los hashes MD5 para palabras comunes, frases, palabras modificadas y strings aleatorios. La **facilidad de crackear un algoritmo MD5** hace posible la existencia de este tipo de tablas.

Generar este tipo de tablas para un **algoritmo complejo** tarda mucho más, pero es posible también. Una medida apropiada es **añadir un salt al hash**. En este contexto, salt es cualquier frase que se añade a la contraseña antes de crear el hash. Usando un salt se gana mucho terreno frente a este tipo de tablas. Se debería **crear una Rainbow Table específica para tu aplicación** y averiguar cual es el salt en tu aplicación.

2. Mejora de los antiguos métodos de encriptación

Primero veamos la funciones básicas de hashing para PHP:

- md5

Calcula un hash con el algoritmo md5. Si se establece _$raw*output* como true se devolverá en raw binario con una longitud de 16. De normal devuelve un hash de 32 caracteres hexadecimal.

- sha1

```
string sha1 (string $str [, bool $raw_output = false ])
```

Calcula un hash con el algoritmo sha1. Si se establece _$raw*output* como true se devolverá en raw binario con una longitud de 20. De normal devuelve un hash de 40 caracteres hexadecimal.

- hash

```
string hash ( string $algo, string $data [, bool $raw_output = false ]
```

La función toma primero el algoritmo que se desea emplear, *$algo*, y después el string que se desea encriptar, *$data*. El algoritmo puede ser **md5**, **sha128**, **sha256**...

Anteriormente, el siguiente código era un **ejemplo de una buena protección de contraseñas**:

```
class Password {
    const SALT = 'EstoEsUnSalt';
    public static function hash($password) {
        return hash('sha512', self::SALT . $password);
    }
    public static function verify($password, $hash) {
        return ($hash == self::hash($password));
    }
}
// Crear la contraseña:
$hash = Password::hash('micontraseña');
// Comprobar la contraseña introducida
if (Password::verify('micontraseña', $hash)) {
    echo 'Contraseña correcta!\n';
} else {
    echo "Contraseña incorrecta!\n";
}
```

Durante mucho tiempo esto ha sido la mejor forma de protegerse, mejor que usar md5. Se usa un algoritmo mucho más complejo como el sha512, y fuerza a todas

- **Se utiliza un salt, pero todas las contraseñas utilizan el mismo**, por lo que si alguien consigue averiguar una contraseña, o el acceso al código fuente donde puede mirar el hash, se puede hacer una Rainbow Table añadiendo el salt descubierto. La solución es crear un salt aleatorio para cada contraseña que se crea, y guardar el salt con la contraseña de forma que después se pueda recuperar.
- **Se utiliza sha512, un complejo algoritmo que viene con PHP**. Sin embargo también puede ser crackeado a un ratio de 46 millones de cálculos por segundo. Aunque es más lento de crackear que **md5** y **sha1**, todavía no es un nivel de seguridad estable. La solución es utilizar algoritmos que son todavía más complejos y emplearlos varias veces. Por ejemplo emplear un algoritmo **sha512** 10 veces consecutivamente reduciría el intento de hackeo considerablemente.

Las dos soluciones ya vienen por defecto con la librería Hash de contraseñas de PHP.

3. La librería Hash de contraseñas de PHP

La extensión Hash de contraseñas crea un password muy complejo, incluyendo la generación de salts aleatorios. En forma más simple se utiliza la función _password*hash()*, con la contraseña que quieres "hashear", y la extensión lo hace directamente. Es necesario facilitar también el algoritmo que se desea emplear. La mejor opción de momento es especificar **PASSWORD_DEFAULT** (se actualiza siempre que se añada un algoritmo nuevo más fuerte), aunque también es posible **PASSWORD_BCRYPT**.

- password_hash

```
string password_hash ( string $password , integer $algo [, array $opti
```

Es compatible con *crypt()* por lo que los hash de contraseñas creados con *crypt()* se pueden usar con _password*hash()*. Las opciones que se admiten son ***salt*** (para proporcionarlo manualmente, pero esta opción ya está obsoleta en PHP 7 por lo que no conviene usarla) y ***cost***, que denota el **coste del algoritmo** a usar (el valor predeterminado es 10).

```
$hash = password_hash('micontraseña', PASSWORD_DEFAULT, [15]);
```

El coste indica cuánto de complejo debe ser el algoritmo y por lo tanto cuánto tardará en generarse el hash. El número se puede considerar como el número de

Para poder verificar los passwords, deberíamos saber el salt que se ha creado. Si se usa _password*hash()* otra vez y se compara con el anterior, se puede ver que son distintos. Cada vez que se llama a la función, se genera un nuevo hash, por lo que la extensión facilita una segunda función: _password*verify()*. Llamando a esta función y pasando la contraseña proporcionada por el usuario, la función devolverá true si coincide con la almacenada:

```
if(password_verify($password, $hash)){
    // Password correcto!
}
```

Ahora la clase que habíamos puesto al principio se puede refactorizar por una mucho más segura:

```
class Password {
    public static function hash($password) {
        return password_hash($password, PASSWORD_DEFAULT, ['cost' => 15]);
    }
    public static function verify($password, $hash) {
        return password_verify($password, $hash);
    }
}
```

4. Cambios en los métodos de encriptación

Usando la **extensión de encriptación de PHP**, tu aplicación estará con los **últimos estándares en seguridad**, aunque hace algunos años de decía que **SHA-1** era lo mejor. Esto significa que cada vez se van actualizando los logaritmos y por tanto la extensión se irá adaptando.

¿Y qué ocurre con las **contraseñas antiguas**? Para eso está la función _password_needs*rehash()*, que detecta si una contraseña almacenada no cumple con las necesidades de seguridad de la aplicación. La razón puede ser que hayas **aumentado la complejidad con *cost***, o que **PHP haya actualizado el algoritmo**. Por esta razón se ha de elegir PASSWORD_DEFAULT, siempre se estará protegido con la opción más segura disponible.

Cuando un usuario se logea, ahora tendríamos una nueva tarea, llamar a _password_needs*rehash()*, que toma parámetros similares a _password*hash()*. Lo que hace la función _password_needs*rehash()* es decirte si el password necesita un rehash. Depende de ti **cuándo generar un nuevo hash de contraseña y guardarlo**, porque la extensión Hash desconoce cómo deseas hacerlo.

realizarse:

```
class User {
    // Opciones de contraseña:
    const HASH = PASSWORD_DEFAULT;
    const COST = 14;
    // Almacenamiento de datos del usuario:
    public $data;
    // Constructor simulado:
    public function __construct() {
        //  Leer los datos de la base de datos almacenados en $data, como
        //  $data->passwordHash  o  $data->username
    }
    // Funcionalidad de guardar los datos simulada:
    public function save() {
        // Guardar los datos de $data en la base de datos
    }
```

Ya hemos construido la base de la clase **user**, ahora vamos a ver el cambio de contraseña y el login:

```
// Permite el cambio de contraseña:
    public function setPassword($password) {
        $this->data->passwordHash = password_hash($password, self::HASH, ['cost' => s
    }
    // Logear un usuario:
    public function login($password) {
        // Primero comprobamos si se ha empleado una contraseña correcta:
        echo "Login: ", $this->data->passwordHash, "\n";
        if (password_verify($password, $this->data->passwordHash)) {
            // Exito, ahora se comprueba si la contraseña necesita un rehash:
            if (password_needs_rehash($this->data->passwordHash, self::HASH, ['cost'
                // Tenemos que hacer rehash en la contraseña y guardarla.  Simplement
                $this->setPassword($password);
                $this->save();
            }
            return true; // O hacer lo necesario para indicar que el usuario se ha lo
        }
        return false;
    }
}
```

Sitio construido con Symfony & Semantic-UI