



Tema 8

Acceso a Bases de Datos

La necesidad de almacenar información se remonta a siglos atrás. Mantener registros de información sobre diferentes aspectos de la vida

cotidiana o históricos, entre otros, propició los primeros avances de comunicación escrita, llegando a crearse con el paso de los años bibliotecas y toda clase de registros. No obstante, a mediados del siglo pasado, con la aparición de los primeros ordenadores, el concepto de almacenamiento de datos dio un giro radical. Esto, no sólo supuso un notable avance en los dispositivos de almacenamiento, sino también en el nacimiento de las base de datos.

Las bases de datos, tal y como las conocemos hoy en día, tienen sus orígenes en la década de los 70 cuando Lawrence J. Ellison, basándose en los trabajos del inglés Edgar Frank Codd sobre sistemas de bases de datos relacionales, desarrolló lo que hoy en día se conoce como Oracle. Los avances producidos en este campo nos han traído hasta la actualidad, donde IBM, Oracle y Microsoft dominan el mercado de bases de datos, proporcionando numerosas herramientas capaces de manipular ingentes cantidades de información.

Los sistemas de gestión de bases de datos modernos nos permiten mantener actualizada la información, facilitan la relación de búsquedas, disminuyen los costes de mantenimiento, implementan sistemas de control de acceso y almacenan las preferencias de los usuarios, entre otras muchas características. Obviamente, no es nuestro objetivo estudiar a fondo la administración y gestión de bases de datos, aunque si nos centraremos en cómo acceder a ellas desde nuestras aplicaciones web.

Cualquier tienda online almacena la información sobre usuarios, productos, etc., en una base de datos. Al igual que estas, otras muchas aplicaciones web hacen uso de estos motores de almacenamiento de información, que será recuperada y gestionada de forma apropiada por la aplicación según sea necesario. Acceder a los datos contenidos en estos almacenes es relativamente sencillo si utilizamos cualquier herramienta diseñada a tal efecto. Pero, ¿cómo podemos hacerlo desde nuestra aplicación web?

PHP nos permite trabajar con prácticamente todos los gestores de bases de datos disponibles hoy en día en el mercado. Nosotros utilizaremos MariaDB, un motor con licencia GPL derivado directamente de MySQL y desarrollado inicialmente por Michael Widenius.

El núcleo de PHP hace uso de diferentes extensiones para ampliar su funcionalidad. Generalmente, cada una de estas extensiones proporciona una API con el objeto de que el desarrollador pueda utilizar sus prestaciones. Recordemos que una interfaz de programación de aplicaciones, o simplemente API, define las clases, métodos funciones y variables que una aplicación necesita para realizar una determinada tarea.

Dividimos las API en dos grupos: **procedimentales** y **orientadas a objetos**. La diferencia entre ambas es bastante obvia. En tanto que las primeras hacen uso directo de funciones, las orientadas a objetos instancian clases proporcionando, a través de diferentes métodos, acceso a las funcionalidades de la API. Actualmente, PHP facilita dos tipos de conexión: **mysqli** y **PDO**; cada una de ellas será objeto de estudio a lo largo de este tema.

Extensión MySQLi

La extensión MySQLi (mysql improved) habilita el acceso a las funcionalidades proporcionadas por MySQL y, obviamente también por MariaDB ya que, como se dijo anteriormente, éste es un *fork* directo del primero. Esta extensión ofrece una interfaz dual, sin embargo, en este libro trabajaremos únicamente con la clase **mysqli**.

Estableciendo una conexión

Imaginemos que nos encontramos desarrollando una aplicación web orientada a la gestión de nuestra biblioteca ya que, como aficionados a la lectura, contamos con una enorme cantidad de libros y deseamos clasificarlos convenientemente. Además de construir de forma apropiada la base de datos que se encargará de contener los datos asociados a cada uno de los libros, la aplicación deberá permitirnos acceder a dicha información para poder gestionarla.

Obviamente, la base de datos se encontrará bajo la tutela de un gestor de datos que, además será accesible desde un servidor. Nuestra aplicación, por tanto, deberá en primer lugar establecer una conexión con dicho servidor, para poder empezar a comunicarse con el motor de bases de datos y ejecutar diferentes órdenes que permitan manipular la información relacionada con nuestra biblioteca.

El constructor de la clase **mysqli** acepta un total de cinco parámetros, de los cuales, al menos serán necesarios tres de ellos: la **dirección** del servidor de bases de datos, el **usuario** y **clave** de acceso al mismo. Cada vez que instanciamos la clase se intentará

establecer una conexión con el servidor indicado. Sin embargo, esto no nos garantiza que dicho enlace se realice con éxito, por lo que siempre es conveniente realizar una comprobación.

```
$mysqli = new mysqli("localhost", "root", "");  
if ($mysqli->connect_errno)  
    die("**Error de conexión: {$mysqli->connect_error}");
```

El fragmento de código anterior intenta establecer una conexión con un servicio de gestión de bases de datos instalado en nuestro propio equipo. El usuario es `root` y, aunque no es para nada seguro, éste no tiene contraseña. Seguidamente, comprobamos si se ha producido algún error de conexión. La propiedad `connect_errno` devuelve un posible código de error y cero si la conexión se ha realizado con éxito. Si se produce un fallo, utilizamos la función `die` para detener la ejecución del script, que mostrará un mensaje con la descripción del error producido y que se encuentra accesible a través de la propiedad `connect_error`.

Además de los valores ya indicados, el constructor de la clase puede recibir opcionalmente otros dos: el nombre de la base de datos a la que queremos acceder, y el puerto de conexión que, generalmente es el **3306**. No obstante, podemos seleccionar la base de datos *a posteriori*.

```
$mysqli->select_db("biblioteca");
```

A partir de este momento, siempre que no se haya producido ningún error en la conexión, tendremos acceso a las tablas de la base de datos elegida para poder gestionar la información que contiene.

Ejecutando sentencias SQL

SQL (Structured Query Language) es un lenguaje de cuarta generación orientado a la gestión de sistemas de bases de datos. Desarrollado en el año 1974 por Donald D. Chamberlin, y basado originalmente en el álgebra y cálculo relacional, pone a disposición del administrador de bases de datos una gran cantidad de herramientas, entre ellas, las que utilizaremos nosotros para la inserción, consulta, actualización y borrado de información.

Será, por tanto, necesario tener conocimientos de este lenguaje para poder ejecutar sentencias SQL sobre nuestra base de datos, para lo cual podremos hacer uso de tres métodos diferentes: **real_query** y **multi_query** y **query**, siendo ésta última la más habitual y en la que nos centraremos en este capítulo. Supongamos que queremos buscar todos los libros que tenemos almacenados en nuestra base de datos.

```
$resultados = $mysqli->query("SELECT * FROM libros ;");
```

Como vemos, el método recibe una cadena con el comando SQL que queremos lanzar contra la base de datos. Se recomienda escapar previamente la cadena para evitar problemas de seguridad, para lo cual se deberá emplear un método apropiado.

```
$sql = $mysqli->real_escape_string("SELECT * FROM libros ;") ;  
$resultados = $mysqli->query($sql) ;
```

La sentencia anterior no presenta ningún problema pero, tal y como se verá al final del capítulo, podríamos encontrarnos con caracteres extraños que, formando parte de la consulta, podrían dar lugar a problemas serios de seguridad. Sea como sea, la función query devolverá:

1. Un objeto de tipo **mysqli_result** si realizamos una consulta de tipo **SELECT**, **SHOW**, **DESCRIBE** o **EXPLAIN**, y ésta fue un éxito.
2. Si realizamos una consulta diferente a las indicadas anteriormente y el resultado de la operación fue un éxito, el método devolverá **true**.
3. El valor booleano **false**, si se ha producido un error.

El método anterior puede recibir opcionalmente un segundo parámetro, permitiéndonos especificar el modo en que se deberá manejar el resultado de la consulta.

Modo MYSQLI_STORE_RESULT

Cuando utilizamos este modo, que precisamente es el definido por defecto para el método query, se estará empleando un método de almacenamiento en buffer. Esto significa que, los resultados obtenidos por la consulta SQL son inmediatamente transferidos a la memoria del proceso PHP que está en ejecución. Esto facilitará un gran número de operaciones sobre los resultados, pero no resulta especialmente óptimo cuando manejamos grandes cantidades de datos.

Volvamos al ejemplo anterior, donde realizamos una consulta sobre la tabla libros, esperando obtener una relación de todos los ejemplares almacenados en la base de datos. Si la consulta **SELECT** se realiza con éxito, el método query devolverá un objeto de tipo **mysqli_result** que guardaremos en la variable **\$resultados**.

Podremos encontrarnos nuevamente con la posibilidad de que la consulta haya fallado. En un principio, quizá intentemos recurrir a los métodos explicados anteriormente, pero éstos tan sólo nos servirán para informarnos sobre errores de conexión.

```
if ($mysqli->errno)  
    die("**Error {$mysqli->errno}: {$mysqli->error}") ;
```

Comprobaremos posibles errores en una consulta a la base de datos a través de los métodos **errno** y **error** de la clase **mysqli**. En esta ocasión, el primero de ellos nos proporciona el código de error, y el segundo una descripción del mismo.

Si hemos realizado correctamente la consulta y no se produce error alguno, tocará recuperar la información; pero, ¿cómo lo hacemos? Existen diferentes métodos que nos ayudarán a ello y, en la siguiente tabla resumimos los que seguramente utilizaremos con más asiduidad.

Función	Descripción
fetch_all([tipo])	Devuelve un array con el conjunto completo de registros obtenidos como resultado de la consulta SQL. El parámetro opcional tipo especifica cómo debe ser el array devuelto; esto es, escalar (por defecto), asociativo, lo que indicaremos con a través de la constante MYSQLI_ASSOC , o ambos, indicándolo con el valor MYSQLI_BOTH .
fetch_assoc()	Devuelve en un array asociativo una fila y la elimina del conjunto de resultados. Devolverá el valor null si no hay más registros en el conjunto de resultados.
fetch_row()	Devuelve en un array escalar una fila y la elimina del conjunto de resultados. Al igual que la anterior, devolverá null si no hay más resultados.
fetch_array([tipo])	Devuelve en un array una fila y la elimina del conjunto de resultados. A través de las constantes MYSQLI_ASSOC , MYSQLI_NUM y MYSQLI_BOTH (valor por defecto) podemos determinar si el array es asociativo, escalar o ambos. Devuelve null si no hay más filas.
fetch_object([clase])	Realiza el mapeo de un registro sobre un objeto. Si no especificamos el nombre de una clase, el objeto será de tipo stdClass . Hemos de tener en cuenta que los nombres de los campos son sensibles a mayúsculas y minúsculas.

Aunque podemos hacer uso de cualquiera de los métodos anteriores, en los ejemplos de este libro utilizaremos únicamente **fetch_object**. Teniendo esto en cuenta, retomemos nuestro ejemplo.

```
$registro = $resultados->fetch_object() ;
```

Tras la consulta, utilizamos **fetch_object** para recuperar el primer registro de entre todos los obtenidos y, éste además se eliminará del conjunto de resultados.

isbn	titulo	autor	editorial
9788417761769	Los Goonies	James Kahn	Duomo
9788420482767	Momo	Michael Ende	Alfaguara
...

isbn: 9788417761769
titulo: Los Goonies
autor: James Kahn
editorial: duomo

stdClass

La operación anterior almacenará en la variable **\$registro** un objeto genérico de tipo **stdClass** (*standard class*) en el que, cada uno de sus miembros corresponde a los diferentes campos del registro. Todas estas propiedades se definen automáticamente como públicas y accedemos a ellas como tal.

```
echo "ISBN: {$registro->isbn}<br/>" ;  
echo "Título: {$registro->titulo}<br/>" ;  
...
```

Obviamente, esto mostrará los valores de un único registro. Lo habitual será obtener más de un resultado, ya que nuestra biblioteca contará con más de un libro, por lo que tendríamos que repetir las operaciones anteriores hasta que no queden registros en el conjunto de resultados.

```
while($registro = $resultados->fetch_object()):  
    echo "ISBN: {$registro->isbn}<br/>" ;  
    echo "Título: {$registro->titulo}<br/>" ;  
    ...  
endwhile ;
```

Cómo desconocemos el número exacto de registros que ha devuelto la consulta, pero sabemos que el método **fetch_object** devolverá null cuando no haya más registros en el *buffer*, será fácil extraer toda la información utilizando un sencillo bucle. Este método puede recibir opcionalmente un parámetro donde le indiquemos el nombre de la clase que debe instanciar.

El uso de objetos de clase **stdClass** nos limita bastante por lo que, generalmente realizaremos el mapeo utilizando clases definidas por nosotros mismos. En este caso, es **importante** recordar que durante el proceso de mapeo, las propiedades se crean antes de la llamada al constructor, por lo que no podríamos utilizar la promoción de propiedades para definir la clase.

MODO MYSQLI_USE_RESULT

Si deseamos hacer uso de este modo, tendremos que indicarlo explícitamente a través de la constante **MYSQLI_USE_RESULT** y, tendremos que saber que estaremos empleando un

método no *bufferado*. Es decir, ejecutada la consulta, el servidor devuelve una referencia o resource al conjunto de datos obtenido, que permanecerán en la memoria del servidor de base de datos a la espera de ser extraídos.

Esto nos limita, no sólo a la hora de operar sobre los datos, sino también cuando queramos hacer cualquier otra consulta ya que, hasta que no hayamos obtenido el conjunto completo de resultados, no podremos realizar más consultas a través de la misma conexión. Aún así, el modo **MYSQLI_USE_RESULT** es el recomendado cuando esperamos obtener un elevado número de resultados. En caso de haber obtenido el resultado que buscábamos, si quisiéramos desechar el resto y liberar la conexión, bastará con hacer una llamada al método **free_result**.

Cerrando una conexión

Cuando hayamos terminado de comunicarnos con la base de datos, si no tenemos pensado realizar más operaciones sobre ella, debemos cerrar la conexión establecida con anterioridad. Utilizamos para ello el método **close** que, además destruirá los resultados almacenados en el *buffer* y que no hayan sido liberados previamente.

El uso de esta función no es obligatorio ya que, al término de un script de PHP, todos los recursos son liberados. Sin embargo, se recomienda utilizarla siempre que sea necesaria para devolver los recursos a PHP y al motor de bases de datos, mejorando de esta manera el rendimiento.

PDO (PHP Data Objects)

Los **PHP Data Objects** (objetos de datos de PHP) proporcionan una capa de abstracción que permitirá a los métodos acceder a los datos, independientemente del gestor de bases de datos utilizado. Si estamos trabajando con una base de datos MySQL y queremos migrar a Oracle, la interfaz de PDO nos permite hacer el cambio sin que nuestro código se vea prácticamente afectado. No obstante, la versatilidad de PDO presenta un pequeño inconveniente: no puede aprovechar las pequeñas diferencias existentes entre los diferentes motores, haciendo que sea menos eficiente que **mysqli** en determinadas situaciones.

Sea como sea, elegir entre una conexión u otra dependerá de las características y necesidades de nuestra aplicación, por lo que tendremos que valorar qué prestaciones presentan ambas conexiones y decantarnos por la más apropiada. **PDO** y **mysqli** comparten ciertas funcionalidades. Sin embargo la que nos ocupa en este apartado, no sólo incorpora soporte para numerosos gestores de bases de datos, sino que además

proporciona una capa de seguridad adicional, así como la posibilidad de emplear sentencias preparadas del lado del cliente; esto es, cuando se prepara la sentencia no necesita comunicación alguna con el servidor de bases de datos.

Conectando al servidor de bases de datos

Cuando utilicemos los PHP data objects estableceremos una conexión con el servidor de bases de datos a través de una instancia de la clase PDO. El constructor necesita una cadena o **dns** (Data Source Name) con toda la información necesaria para conectarse a la base de datos y, opcionalmente el nombre de usuario, contraseña y un *array* asociativo con opciones específicas del controlador

```
$pdo = new PDO("mysql:host=localhost;dbname=biblioteca;charset=utf8",  
               "usuario","contraseña") ;
```

La línea de código anterior devolverá un objeto PDO que facilitará la comunicación con el gestor de bases de datos. Los parámetros de configuración que se deben emplear en la cadena **dns** varían de un controlador a otro, por lo que será necesario recurrir a la documentación para conocerlos. Se utilizará **mysql**, **pgsql**, **sqlite**, **firebird**, **informix** u **oci** como prefijo del **dns** según deseemos establecer una conexión con motores MySQL, PostgreSQL, SQLite, Firebird, Informix u Oracle respectivamente.

Si se produce algún error en la conexión se lanzará una excepción de tipo **PDOException**. Aunque podemos dejar que el manejador global de excepciones resuelva la situación, generalmente es recomendable que seamos nosotros quienes manejamos la excepción convenientemente.

```
try {  
    $pdo = new PDO("mysql:host=localhost;dbname=biblioteca", "usuario",  
                  "contraseña") ;  
} catch (PDOException $excep) {  
    die("**Error: ".$excep->getMessage()) ;  
}
```

En caso de error, el proceso finalizará mostrando un mensaje de error; en otro caso, si la conexión se realiza con éxito, permanecerá activa hasta que se cierre, para lo cual tendremos que destruir el objeto asegurándose de haber eliminado todas las referencias al mismo.

```
$pdo = null ;
```


Consultas SQL

Las consultas de selección de información se realizan de forma idéntica a cómo lo hacíamos utilizando la extensión **mysqli**, obteniendo en este caso un objeto de clase **PDOStatement** o **false** en caso de error.

```
$resultado = $pdo->query("SELECT * FROM libros ;") ;
```

Hemos de tener en cuenta que, para evitar problemas de seguridad, antes de lanzar la *query* debemos escapar correctamente la cadena con la consulta SQL. Se emplea para ello el método **quote** de la clase **PDO** que, en función del controlador utilizado (algunos no soportan esta funcionalidad), escapará la cadena convenientemente.

```
$sql = $pdo->quote("SELECT * FROM libros;" ) ;
```

Este método devolverá la cadena escapada o **false** si el driver no soporta el escapado. Obviamente, para poder emplear esta función deberemos definir el conjunto de caracteres con el que vamos a trabajar. Sea como sea, la documentación nos recomienda encarecidamente utilizar lo mínimo posible este método en beneficio de las sentencias preparadas que estudiamos en el siguiente apartado.

```
$resultado = $pdo->query($sql) ;
```

Tras lanzar la consulta, obtendremos un objeto de tipo **PDOStatement** que contiene, además de otra información, el total de registros de la tabla libros. Internamente, **PDO** utiliza cursores para recorrer los registros obtenidos como resultado de una consulta; es más, **SELECT** es el único comando susceptible de utilizar este mecanismo.

```
foreach($resultado as $item):  
    echo "ISBN: {$item["isbn"]}<br/>" ;  
    echo "Título: {$item["titulo"]}<br/>" ;  
    ...  
endforeach ;
```

Si tras la consulta no accedemos a todos los resultados, será obligatorio liberar los recursos asociados al objeto antes de realizar otra operación.

```
$resultado->closeCursor() ;
```

La documentación oficial de PHP nos recomienda evitar el método **query** cuando trabajemos con **PDO** y siempre que nos sea posible. Esto no significa que no podamos utilizarlo, es más, suele emplearse para realizar consultas de tipo **SELECT**, pero tendremos

que hacerlo siempre con cuidado y en situaciones que no sean susceptibles a problemas de seguridad.

Entonces, si no empleamos el método anterior, ¿cómo realizamos una consulta con **PDO**? El método **execute**, que además presenta un mayor rendimiento, nos ayudará a realizar cualquier tipo de consultas SQL aunque suele utilizarse con más frecuencia con comandos como **INSERT**, **DELETE** o **UPDATE**.

Sin embargo, lanzar una consulta con **execute** requiere que ésta haya sido preparada con anterioridad. La mayoría de los motores de bases de datos actuales admiten **sentencias preparadas**, que pueden considerarse como una mera plantilla SQL.

Imaginemos que queremos insertar un determinado libro en la base de datos. Construimos la plantilla SQL utilizando tantos marcadores de parámetros como sean necesarios, que podrán ser definidos de dos formas diferentes.

- a. **Etiquetas**. Cada marcador será una etiqueta o identificador que deberá comenzar obligatoriamente con dos puntos, tal y como vemos en el siguiente ejemplo.

```
$sqlp = $pdo->prepare("INSERT INTO libros VALUES (:isbn, :tit, :aut, :edi) ;") ;
```

- b. **Comodín**. En esta ocasión, los marcadores de parámetros quedarán definidos a través del símbolo **?** que, posteriormente será sustituido por los valores correspondientes.

```
$sqlp = $pdo->prepare("INSERT INTO libros VALUES (?, ?, ?, ?) ;") ;
```

Definida la plantilla, y antes de lanzar la consulta contra la base de datos, sólo nos quedará asociar cada marcador con el valor correspondiente, para lo cual tenemos a nuestra disposición dos métodos: **bindParam** y **bindValue**. La interfaz de ambos es similar, aunque difieren en su funcionamiento.

Vinculación de valores

La vinculación de valores se realizará a través del método **bindValue**, que necesita obligatoriamente dos parámetros: la **etiqueta** y el **valor** asociado a la misma; el tercer parámetro es optativo

```
$sqlp->bindValue(":isbn", "9788445000656", PDO::PARAM_STR) ;
```

Como vemos, asociamos a la etiqueta **:isbn** el valor **9788445000656** que, además deberá ser de tipo string. El tipo del parámetro se indicará utilizando las constantes definidas por la clase **PDO** a tal efecto; en este caso hemos utilizado **PDO::PARAM_STR**.

Si preferimos el uso de comodines, el primer parámetro deberá ser un valor numérico que represente la posición del comodín dentro de la plantilla, teniendo en cuenta que el primero de dichos comodines ocupará siempre la posición cero.

```
$sqlp->bindValue(0, "9788445000656", PDO::PARAM_STR) ;  
$sqlp->bindValue(1, "El Hobbit", PDO::PARAM_STR) ;
```

Aunque en los ejemplos anteriores hemos utilizado valores constantes para asociarlos las etiquetas o comodines correspondientes, también podríamos hacer uso de variables e incluso podríamos omitir el último parámetro, si así lo deseamos.

```
$sqlp->bindValue(":isbn", $isbn) ;
```

Vinculación de parámetros

Utilizamos en esta ocasión el método **bindParam** para asociar etiquetas o comodines con los valores correspondientes. Existen varias diferencias entre este método y el anterior.

```
$sqlp->bindParam(":isbn", $isbn, PDO::PARAM_STR, 13) ;
```

Excepto en el nombre del método utilizado, la línea de código anterior no difiere en nada con respecto a ejemplos previos. Sin embargo, sí existe una diferencia subyacente: **bindParam** obliga a que el segundo parámetro sea siempre una variable. Esto es debido a la manera en que funciona el método. La variable se vincula al marcador elegido como una referencia y será evaluada únicamente cuando se ejecute la sentencia. Esto es, podemos cambiar el valor del parámetro tras la vinculación, modificando la variable, ya que ésta se evaluará durante la ejecución de la sentencia.

Ejecutando la consulta

Creada la plantilla y vinculados los parámetros con los valores que correspondan, sólo nos quedará lanzar la consulta SQL. Escribamos el código completo para nuestro ejemplo.

```
$sqlp = $pdo->prepare("INSERT INTO libros VALUES (:isbn, :tit, :aut, :edi)");  
$sqlp->bindValue(":isbn", "9788445000656", PDO::PARAM_STR) ;  
$sqlp->bindValue(":tit", "El Hobbit", PDO::PARAM_STR) ;  
$sqlp->bindValue(":aut", "JRR. Tolkien", PDO::PARAM_STR) ;  
$sqlp->bindValue(":edi", "debolillo", PDO::PARAM_STR) ;  
$sqlp->execute() ;
```

Como puedes ver, el método **execute**, sin más, será el encargado de ejecutar la consulta. Sin embargo, este presenta una característica que, en algunas ocasiones podrá resultarnos de especial utilidad: opcionalmente acepta un *array* escalar o asociativo, que vincula los comodines o etiquetas, respectivamente, con los valores que correspondan. Aunque esto

puede simplificar nuestro código, no siempre es recomendable ya que, los métodos **bindValue** y **bindParam** estudiados anteriormente, añaden una capa de seguridad protegiendo contra inyecciones SQL.













```
$sqlp->execute([ "9788445000656", "El Hobbit", "JRR.Tolkien", "debolsillo" ]) ;
```

Inyección SQL

Imagina que acabas de desarrollar una aplicación web que muestra en pantalla un formulario de acceso. Cuando el usuario introduce sus datos, se realizará una consulta a la base de datos para comprobar si sus credenciales de acceso son correctas, en cuyo caso se le permitirá acceder a la sección privada de la aplicación. Sin embargo, una consulta SQL puede ser manipulada, eludiendo controles de accesos así como las comprobaciones de autenticación y autorización.

Se conoce como **inyección de código SQL** a la técnica con la que un atacante altera comandos SQL para manipular o exponer datos ocultos. Además de éste método, existen otras técnicas de ataque, pero el objetivo de este libro no es ahondar en este tema. Dedicaremos este apartado a introducir el método de explotación más habitual y alguna técnica de evitación que podrás tener en cuenta a la hora de diseñar tus aplicaciones.

Continuemos con la aplicación de gestión de nuestra biblioteca. Añadimos ahora una pantalla de login donde, cada usuario introducirá usuario y contraseña para poder acceder a su cuenta. Añadimos a nuestra base de datos biblioteca una tabla llamada usuarios y añadimos un par de registros con datos de usuarios, tal y como vemos en la siguiente imagen.

		usr	pass	nombre	apellidos	direccion	ciudad
<input type="checkbox"/>	 Editar	 Copiar	 Borrar	mike@goon.com	mikey	Michael Walsh	Muelles de Goon Astoria
<input type="checkbox"/>	 Editar	 Copiar	 Borrar	data@goon.com	data	Richard Wang	Muelles de Goon Astoria
<input type="checkbox"/>	 Editar	 Copiar	 Borrar	gordi@goon.com	gordi	Lawrence Cohen	Muelles de Goon Astoria
<input type="checkbox"/>	 Editar	 Copiar	 Borrar	bocazas@goon.com	bocazas	Clarke Devereaux	Muelles de Goon Astoria

Utilizaremos la extensión **mysqli** para construir el mecanismo de fogueo realizando una simple consulta en la base de datos.

```
$sqli = new mysqli("localhost", "root", "");  
$sqli->select_db("biblioteca");  
$sqli->query("SELECT * FROM usuarios WHERE usr='$usr' AND pass='$pass' ;") ;
```

Suponemos que las variables **\$usr** y **\$pass** tienen los valores introducidos por el usuario a través del formulario. Accedemos ahora nuestra aplicación y ésta nos muestra el formulario.

¿Podríamos acceder sin necesidad de contraseña? Si conocemos el nombre de algún usuario, y teniendo en cuenta la sentencia SQL anterior, podríamos foguearnos fácilmente sin necesidad de conocer su contraseña.

Usuario:

Contraseña:

Hemos dejado visible el campo de contraseña para que podamos ver exactamente qué estamos haciendo. Como dijimos anteriormente, conocemos el nombre de usuario, pero no su contraseña. Escribimos en el campo usuario el DNI de dicho usuario y como contraseña cualquier valor. Al pulsar el botón **Entrar**, estos datos se enviarán al script que procesa la información y lanzará la consulta SQL. Pero, ¿qué consulta exactamente? Sabiendo que las variables **\$usr** y **\$pass** contienen los valores enviados a través del formulario, así que los sustituimos en la consulta.

```
SELECT * FROM usuarios
WHERE usr='mike@goon.com' #AND pass='cualquiercosa' ;
```

El resultado de ejecutar esta consulta provocará el acceso inmediato del usuario a la cuenta asociada al email *mike@goon.com*. ¿Qué ha sucedido? Sencillamente, hemos consultado únicamente si el usuario existe en la base de datos, convirtiendo en un comentario el resto de la consulta. Recordemos que MariaDB utiliza el símbolo **#** para introducir comentarios en el código.

Conocíamos el nombre de usuario de Mike Walsh y esto nos ha garantizado el acceso sin conocer su contraseña. Sin embargo, vayamos más allá. Imagina ahora que no conocemos ni nombre de usuario, ni contraseña.

Usuario:

Contraseña:

Tal y como sucedió antes, la sentencia SQL que se lanzará sobre la base de datos será la siguiente.

```
SELECT * FROM usuarios
WHERE usr='' OR 1=1 #AND pass='cualquiercosa'
```

El motor de bases de datos comprobaría si existe un registro cuyo nombre de usuario sea vacío, lo cual no ocurre, por lo que dicha comprobación devolverá **falso**. Sin embargo, la siguiente condición es siempre cierta, ya que **1** es siempre igual a **1**. Como estamos utilizando un **OR** para relacionar ambas condiciones, basta con que una de éstas sea cierta para que la consulta se ejecute correctamente y se devolverán todos los registros existentes en la tabla usuarios.

Un ataque de inyección de código consta básicamente de dos fases. La primera de ellas consiste en investigar la respuesta de la aplicación cuando le proporcionamos valores inesperados. En función de los resultados obtenidos y la observación, llevamos a cabo el ataque a través de una entrada cuidadosamente planificada, que será interpretada como parte de una sentencia SQL.

Descubrir estas vulnerabilidades es el primer paso para defendernos contra este tipo de ataques. Debemos sestejar concienzudamente nuestro código, proporcionando valores insospechados y comprobando que responde de manera adecuada. Si descubrimos una falla de seguridad, tendremos que remediarla inmediatamente. Generalmente, si estamos haciendo uso de la extensión **mysqli** bastará con escapar las cadenas, filtrándolas previamente con el método **real_escape_string**.

```
$usr = $sqli->real_escape_string($_POST["usr"]);  
$pass = $sqli->real_escape_string($_POST["pass"]);  
$sqli->query("SELECT * FROM usuarios WHERE usr='$usr' AND pass='$pass' ;");
```

Escapar las cadenas da como resultado la siguiente sentencia SQL.

```
SELECT * FROM usuarios  
WHERE usr='\' OR 1=1 #' AND pass='cualquiercosa');
```

El método **real_escape_string** ha escapado el carácter comilla (') que habíamos escrito en el campo usuario, por lo que se interpretará como un carácter más y no como un elemento propio de la sintaxis de SQL.

Suele ser habitual codificar la contraseña utilizando algoritmos de cifrado como **MD5**, **SHA/SHA1** o **AES**. Estos algoritmos son de cifrado unidireccional, esto es, una cadena codificada no podrá ser revertida. Supongamos que utilizamos el primero de ellos. En primer lugar, al construir nuestra base de datos, el campo pass deberá tener una longitud mínima de 32 caracteres. Almacenadas las contraseñas convenientemente bajo este sistema de encriptación, antes de realizar la consulta a la base de datos, además de escapar la cadena tendremos que codificarla utilizando la función **md5**.

No obstante, este método de encriptación, aunque es el más simple, no es el más recomendado. Los algoritmos **SHA** y **SHA1** son sinónimos y, mucho más seguros que **MD5**.

Si optamos por almacenar información codificada con alguno de estos métodos, necesitaremos que el campo de la base de datos tenga una longitud mínima de 40 caracteres.

El más seguro de los algoritmos criptográficos que hemos mencionado es **AES**. Este método precisa de una llave privada, necesaria para encriptar y desencriptar cualquier información. Necesitaremos en la base de datos un campo de tipo **BLOB** para almacenar información codificada mediante **AES**.

La extensión **PDO** de PHP, junto al uso de sentencias preparadas, añade una capa de seguridad adicional que previene problemas de inyección de código. Obviamente, toda precaución es poca si no nos aseguramos de proteger la base de datos convenientemente y, no sólo utilizamos el usuario root, sino que además concedemos todos los privilegios al usuario con el que se realiza la conexión al servidor de bases de datos.

Obviamente, el atacante debe tener conocimientos suficientes para poder realizar su intrusión con éxito. Está en nuestra mano proporcionar una mínima información que pueda proporcionar cualquier tipo de pista. En resumen:

- a. La seguridad empieza por la base de datos, de manera que tendremos que asegurar que nuestra aplicación no se conecte al servidor como superusuario o propietario. Se establecerá una conexión a través de **usuarios con privilegios limitados**.
- b. Aunque sea redundar en lo que ya dijimos anteriormente, en la medida de lo posible, se recomienda utilizar sentencias preparadas que permiten **comprobar el tipo de los datos de entrada**.
- c. Debemos utilizar **cadenas bien escapadas**, utilizando para ello las funciones recomendadas según la API utilizada en cada caso.
- d. Y, sobre todo, aunque se recomienda hacerlo durante el proceso de depuración, cuando la aplicación esté en producción **no deberá mostrarse información relevante**, como mensajes de error que desvelen el nombre de tablas o campos de la base de datos. Todo esto puede proporcionar pistas a un posible atacante.

Bibliografía

PHP.net

Guía oficial del lenguaje

PHP and MySQL Web Development
Luke Welling, Laura Thompson
Developer's Library

Learning PHP7
Antonio López
Packt Publishing