

## IMPORTANCIA DE LOS PREPARED STATEMENTS

### ATAQUES SQL INJECTION Y BLIND INJECTION



---

#### ¿Qué son los Prepared Statements?

---

Los Prepared Statements permiten separar los datos que deben ser pasados a una consulta SQL de la cadena la consulta.

Es decir, en lugar de escribir:

```
Select nombre,apellido from clientes where id_cliente='05940329';
```

Usaremos:

```
PREPARE stmt FROM "Select nombre,apellido from clientes where id_cliente= ?";
```

Es decir, enviamos la cadena de consulta al motor de base de datos sin los datos.

---

#### ¿Por qué es importante usar los Prepared Statements?

---

Los Prepared Statements colaboran en la prevención de los ataques conocidos como **inyecciones SQL o SQL injection**. Al pasar los datos separados de la cadena de la consulta no se podrá alterar los datos de la cadena.

En MySQL 4.1 o superior definiremos:

```
SET @id_cliente = "05940329";
```

Y ejecutamos la consulta con:

```
EXECUTE stmt USING @data;
```

En aplicaciones web "open source" ocurre muy seguido este tipo de ataques. Si instala un foro tipo PHPBB cuyo código fuente es conocido por otros programadores malintencionados, puede sufrir fácilmente un ataque por inyección de SQL dado que sin importar que el parámetro se pasado por el método GET o POST, el programador conocerá dónde y cómo recibirlo, en qué comando SQL aplicarlo y cuál será el resultado. Así podrán alterar las cadenas que contienen los INSERTS para generar respuestas y temas con cualquier contenido en los foros.

Inclusive se podría alterar una consulta SQL para que en lugar que devolver los datos de un cliente específico, devuelva los datos de todos los clientes si la consulta es de la forma:

```
Select * from clientes where nombre like "%xxxxx%"
```

Si "xxxxx" proviene de una variable recibida por GET o POST.

Una forma de evitar este tipo de ataques sería validar que el contenido de “xxxxx” no contenga caracteres especiales como “%”, “?” u otro distinto a los caracteres válidos para el campo en cuestión. En caso que el usuario necesite ingresar ciertos caracteres especiales habría que aplicar la función `mysql_escape_string` para que estos caracteres sean tomados como simples caracteres y no como parte de la consulta.

Adicionalmente, es recomendable que la verificación de los datos no sólo se produzca del lado del cliente mediante funciones Javascript si no también en las otras capas de la aplicación web y no mostrar los mensajes de error del motor de base de datos que podrían ser usados por un atacante para ejecutar un **blind injection**.

---

## Ataques Blind Injection

---

Sobre la misma cadena de ejemplo:

**Select nombre,apellido from clientes where id\_cliente='05940329';**

Si la aplicación PHP o JSP recibe como parámetro el ID de cliente tendríamos algo como esto:

```
$x_idcliente= $_GET["p_cliente"];  
Select nombre,apellido from clientes where id_cliente="'.$x_idcliente.'";
```

Y la llamada a la página de la aplicación será:

**[http://www.btrew.com/app/cclientes.php?p\\_cliente=05940329](http://www.btrew.com/app/cclientes.php?p_cliente=05940329)** <sup>(1)</sup>

Si en la línea de URLs del navegador alteramos esta llamada por:

**[http://www.btrew.com/app/cclientes.php?p\\_cliente=05940329 and 5=5](http://www.btrew.com/app/cclientes.php?p_cliente=05940329 and 5=5)** <sup>(2)</sup>

“5=5” o cualquier otra expression booleana que nos devuelva un valor verdadero y la aplicación nos devuelve el mismo resultado en <sup>(1)</sup> y <sup>(2)</sup>, entonces la aplicación es vulnerable a ataques por Sql ciegos (Blind attacks). Los ataques a ciegas se basan en la prueba-error, es decir, ir alterando la cadena Sql para ir obteniendo datos.

Por ejemplo, para averiguar cómo se llama la tabla de clientes podríamos ir probando con algo como esto:

**[http://www.btrew.com/app/cclientes.php?p\\_cliente=05940329 and \(Select \\* from clientes\)](http://www.btrew.com/app/cclientes.php?p_cliente=05940329 and (Select * from clientes))**

Si la tabla de clientes efectivamente se llama “clientes” la consulta devolverá el mismo resultado verdadero, si no sabremos que la tabla tiene otro nombre. Si acertamos con el nombre, digamos “tb\_cliente”, la siguiente consulta devolverá verdadero.

**[http://www.btrew.com/app/cclientes.php?p\\_cliente=05940329 and \(Select \\* from tb\\_cliente\)](http://www.btrew.com/app/cclientes.php?p_cliente=05940329 and (Select * from tb_cliente))**

Ahora podríamos averiguar el nombre del campo “clave” con:

**[http://www.btrew.com/app/cclientes.php?p\\_cliente=05940329 and \(Select password from tb\\_cliente\)](http://www.btrew.com/app/cclientes.php?p_cliente=05940329 and (Select password from tb_cliente))**

Si el campo se llama “password” obtendremos un valor verdadero. De la misma forma podríamos averiguar la longitud de la clave:

**http://www.btrew.com/app/cclientes.php?p\_cliente=05940329 and (Select length(password) from tb\_cliente where id\_cliente='05940329') > 6**

Si la consulta devuelve un valor verdadero, entonces la clave tiene más de 6 caracteres. Así podremos ir preguntando si tiene entre 6 y 8, o si es 7 o si es 9 hasta obtener verdadero.

Ahora, aplicando un simple script PHP podríamos “conseguir” la clave, probando con consultas como:

**http://www.btrew.com/app/cclientes.php?p\_cliente=05940329 and ascii(substring(Select password from tb\_cliente where id\_cliente='05940329'),1,1)=65**

Si esta consulta devuelve verdadero, entonces la clave empieza con “A” (ascii 65).

Siendo un poco más creativos podríamos obtener la clave provocando un error que nos devuelva la clave:

**SELECT password FROM tb\_cliente where id\_cliente='05940329'  
UNION SELECT MIN>Password) FROM tb\_cliente where id\_cliente='05940329'**

Esto podría dar un error como este:

**Syntax error converting the varchar value 'juan3458' to a column of data type integer.**

Donde “juan3458” es la clave que buscamos. De manera similar y luego de haber “averiguado” los campos de la tabla podríamos “insertar” un nuevo usuario o cliente para “ingresar” de manera más rápida.

Otro punto importante son los privilegios con el usuario que accesa por internet tiene sobre la base de datos. Si tienen amplios permisos o privilegio de administrador se puede ir mucho más allá dado que motores como SQL Server presentan store procedures que se pueden intentar ejecutar con esta técnica de ataque. Por ejemplo, sp\_OACreate que crea una instancia de un objeto OLE (ActiveX / COM). Si se permite un servidor OLE de proceso interno se tiene acceso a la memoria y a otros recursos que posea SQL Server. “Un servidor OLE de proceso interno puede dañar la memoria o los recursos de SQL Server, y causar resultados imprevisibles, como una infracción de acceso a SQL Server.”<sup>1</sup>

También es posible intentar ejecutar xp\_cmdshell sobre SQL Server para ejecutar directamente comandos del sistema como: “ EXEC xp\_cmdshell 'net stop sqlserver' ” para detener el servicio. Sin embargo, de forma predeterminada, la opción xp\_cmdshell está deshabilitada en las instalaciones nuevas para SQL Server 2005 o superior.

Para prevenir un ataque SQL Injection o Blind Injection y para realizar pruebas de vulnerabilidad debemos recordar que cualquier aplicación que permita una entrada que sirva de parámetro para una consulta SQL es vulnerable a estos tipos de ataque.

---

<sup>1</sup> <http://msdn.microsoft.com/es-es/library/ms189763.aspx>