

# ¿Quién diría que un ‘1 = 1’ podría ser tan dañino?

Pablo Gerardo González López (pablog@ciencias.unam.mx)

Mayo, 2019

## Resumen

Los ataques de inyección de código *SQL* son de las principales amenazas cibernéticas. En este artículo se describe en qué consiste este ataque, cómo se produce y algunas recomendaciones de cómo evitarlo.

En mayo del 2011, un grupo de hackers conocidos como *LulzSec* hacían la siguiente declaración en su sitio web:

*Our goal here is not to come across as master hackers, hence what we're about to reveal: SonyPictures.com was owned by a very simple SQL injection, one of the most primitive and common vulnerabilities, as we should all know by now. From a single injection, we accessed EVERYTHING. Why do you put such faith in a company that allows itself to become open to these simple attacks?*

*What's worse is that every bit of data we took wasn't encrypted. Sony stored over 1,000,000 passwords of its customers in plaintext, which means it's just a matter of taking it. This is disgraceful and insecure: they were asking for it. [2]*

*Nuestro objetivo aquí no es dejarnos ver como hackers expertos, de lo que se trata es de revelar que: SonyPictures.com quedó a nuestra merced por una simple inyección [de código] SQL, una de las vulnerabilidades más primitivas y comunes, que todos deberíamos conocer a estas alturas. A partir de una sola inyección [de código SQL], accedimos a TODO. ¿Por qué poner tu confianza en una empresa que baja la guardia ante estos simples ataques?*

*Lo que es peor es que cada bit de datos que tomamos no estaba cifrado. Sony almacenaba más de 1,000,000 de contraseñas de sus clientes en texto plano, lo que significa que sólo es cuestión de tomarlas. Esto es vergonzoso e inseguro: ellos lo estaban pidiendo.*

El grupo de hackers había logrado obtener información personal, incluidas contraseñas, direcciones de correo electrónico, direcciones particulares, y fechas de nacimiento de alrededor de un millón de usuarios de *Sony*, además de las credenciales de acceso al sistema de los administradores y 3.5 millones de cupones de música.

*Sony* no fue la primera víctima de este grupo, en ese mismo año también habían logrado comprometer la información de *PBS*, *Fox*, y los cajeros *ATM* del Reino Unido. Su objetivo era evidenciar a nivel mundial lo poco que invertían en seguridad las grandes multinacionales para proteger los datos que almacenaban. Según la declaración, bastó con utilizar un único ataque de inyección de código *SQL* para conseguir toda la información de la base de datos, la cual fue publicada más tarde para que pudiera ser accedida por cualquiera en la web.

A pesar de que esta historia tiene casi 10 años de haber sucedido<sup>1</sup>, los ataques de inyección de código *SQL* siguen siendo una de las principales amenazas cibernéticas<sup>2</sup>, de modo que no es necesario ser una multinacional para ser blanco de este ataque. Hoy en día casi cualquiera que tenga conocimientos básicos de computación puede poner en marcha un servidor web, ya sea para recabar o compartir información, sin embargo pocos son conscientes de la existencia de este tipo de ataque por lo que es necesario conocer en qué consiste para poder estar prevenidos.

## *SQL Injection*

Una inyección de código *SQL*, en inglés *SQL Injection*, «es un tipo de ciberataque en el que un hacker usa un trozo de código *SQL* (Lenguaje de consulta estructurado) para manipular una base de datos y acceder a información potencialmente valiosa» [1]. Basta con aprovecharse de un error en la programación del servidor para obtener el acceso completo a la base de datos que suministra a un sitio web.

*SQL* es el lenguaje declarativo que los sistemas manejadores de bases de datos implementan con el fin de administrar, recuperar o realizar cambios en la información almacenada en una base de datos. Aunque cada sistema de bases de datos implementa su propia versión de este lenguaje, existe un conjunto de operaciones estándar que se basan en el álgebra y cálculo relacional.

Generalmente, los usuarios interactúan con un sitio web a través de formularios. El servidor toma parámetros de consultas en *SQL*, previamente codificadas, que vienen incrustados en las solicitudes de los usuarios (ya sea en las *URL*'s o dentro de las peticiones del protocolo *HTTP*). Después solicita a la base de datos que se ejecuten, ésta, a continuación, se encarga de obtener el resultado de dichas consultas y devolverlas al servidor web para que las formatee y envíe

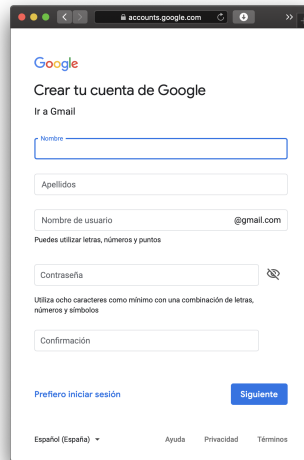
---

<sup>1</sup>Noticia del *Huffpost* sobre el ataque a *Sony*. [https://www.huffpost.com/entry/sony-pictures-hacked-lulzsec\\_n\\_870615](https://www.huffpost.com/entry/sony-pictures-hacked-lulzsec_n_870615)

<sup>2</sup>Infografía del Parlamento Europeo sobre Ciberseguridad. <http://www.europarl.europa.eu/news/en/headlines/security/20190307ST030713/meps-work-to-boost-europe-s-cyber-security-infographic>

de vuelta al usuario. Al incrustar una consulta dolosa en la dirección de una página web en particular, los hackers buscan extraer, sin autorización, información de la configuración de la base de datos con el fin de hacerse una idea de cómo es su estructura interna para así refinar sus consultas y procesar consultas no permitidas, o incluso ejecutar instrucciones para insertar nueva información en lugar de recuperarla.

Figura 1: Ejemplo de formulario de registro de *Google*.

A screenshot of the Google account creation page. The page is titled "Crear tu cuenta de Google" and "Ir a Gmail". It contains several input fields: "Nombre", "Apellidos", "Nombre de usuario" (with a placeholder "@gmail.com"), "Contraseña" (with a strength indicator), and "Confirmación". Below the fields are links for "Prefero iniciar sesión" and a blue "Siguiente" button. At the bottom, there are links for "Español (España)", "Ayuda", "Privacidad", and "Términos".

En el caso de *Sony*, los hackers encontraron la combinación correcta para obtener la información de sus clientes a partir de una consulta a los detalles de una de sus películas más populares (Los Caza-fantasmas).

A continuación, se presentan un par de los ejemplos más comunes de dónde se puede encontrar una vulnerabilidad a este ataque, inspirados en las técnicas propuestas por la *OWASP* [3] para detectar problemas de seguridad en aplicaciones web.

## Ejemplos clásicos

### Inicio de sesión

Supongamos que trabajamos en un sitio web que requiere que cada usuario tenga una cuenta propia para realizar acciones dentro del sistema. Para poder identificarlos, pediremos al inicio de la sesión, un nombre de usuario y contraseña que cotejaremos con una base de datos que ya tiene almacenadas las credenciales de acceso al sistema para cada usuario. Para hacer esto último, consideremos la siguiente consulta en *SQL*:

```
SELECT * FROM Users WHERE
    Username='$username'
AND
    Password='$password'
```

Esta es una de las formas más comunes con la que un sitio web autentica a un usuario. Si la consulta devuelve algún registro, significa que dentro de la base de datos existe un usuario con esas credenciales, entonces se le permite iniciar sesión dentro del sitio. En cualquier otro caso, se le niega el acceso.

Si el usuario inserta los siguientes valores en el formulario correspondiente:

```
$username = 1' OR '1' = '1'
$password = 1' OR '1' = '1'
```

La consulta resultará en:

```
SELECT * FROM Users WHERE
    Username='1' OR '1' = '1'
AND
    Password='1' OR '1' = '1'
```

Analizándola, notaremos que la consulta devolverá al menos un valor, pues la condición siempre es verdadera (`OR '1' = '1'`). De esta manera, si el sistema no cuenta con alguna protección, los datos introducidos serán válidos y se habrá autenticado a un usuario dentro del sitio sin saber el nombre de usuario ni la contraseña.

## Consulta de productos en una tienda en línea

Ahora supongamos que tenemos un sitio web que permite consultar el inventario de una tienda en línea. Para poder realizar la búsqueda de un producto podemos considerar la siguiente consulta en *SQL*:

```
SELECT * FROM Products WHERE name='$name_product'
```

Suponiendo que los valores de los parámetros del formulario se envían al servidor a través de una petición del método *GET*<sup>3</sup>, y si el dominio del sitio web vulnerable fuera *www.example.com*, podríamos realizar la siguiente solicitud:

```
http://www.example.com/search?name_product=patinete
```

Cuando el usuario ingresa algún valor de búsqueda (por ejemplo, patinete en este caso) el servidor devolverá una lista de productos con su nombre, descripción, y precio, cuyos nombres coincidan con dicho valor. Ahora, si modificamos el valor de `name_product` por el siguiente:

<sup>3</sup>Descripción del método *GET* del protocolo *HTTP*. <https://tools.ietf.org/html/rfc7231#section-4.3.1>

```
$name_product = patinete' UNION ALL SELECT Name, CreditCardNumber  
, ExpirationDate FROM Customers WHERE '1' = '1'
```

Obtendremos la siguiente consulta:

```
SELECT * FROM products WHERE name='patinete'  
UNION ALL  
SELECT Name, CreditCardNumber, ExpirationDate FROM Customers  
WHERE '1' = '1'
```

Esta última, unirá el resultado de la consulta original con los datos crediticios de la tabla de clientes (`Customers`). Cabe mencionar que en este punto ya se debió haber descubierto cómo está estructurada internamente la base de datos.

De modo que la siguiente solicitud devolverá la lista de productos cuyos nombres sean patinete, seguido de toda la información crediticia de los clientes del sitio web:

```
http://www.example.com/search?name_product=patinete' UNION ALL  
SELECT Name, CreditCardNumber, ExpirationDate FROM Customers  
WHERE '1' = '1'
```

## ¿Cómo evitar este tipo de ataques?

Desde las más básicas a las más complejas, existen diversas formas de proteger un sitio web de este ataque.

Como medidas primordiales se debe evitar que la base de datos regrese mensajes de error al servidor del sitio web. Los mensajes se deberían escribir en un fichero al cual sólo un número restringido de usuarios deberían tener acceso. Además de utilizar los privilegios apropiados para que el servidor web realice consultas a la base de datos, por ejemplo, el código que maneja la página de inicio de sesión debería consultar la base de datos utilizando una cuenta que esté limitada sólo a realizar consultas a las credenciales de acceso, sin posibilidad de realizar modificaciones o inserciones de datos. También, se debería considerar almacenar las contraseñas y cualquier otra información confidencial de forma cifrada.

Existen otras técnicas que manipulan las entradas del usuario con el fin de evitar que se realice una acción diferente a la cual fue programada la consulta original, como las que se mencionan a continuación:

### Reemplazo de caracteres especiales

Un primer método para prevenir la inyección de código *SQL* es incluir un 'carácter de escape' antes de los caracteres especiales introducidos. Para esto, basta con crear una función similar a la siguiente que se ejecute en el servidor antes de enviar la información introducida por el usuario a la base de datos.

```
Function purify(String input) {
    Return replace(input, ['\\', '\0', '\n', '\r', '"', "'", '\x1a
        '], ['\\\\', '\\0', '\\n', '\\r', "\"", '\"', '\\Z'])
}
```

La función `purify()` tomará la entrada y agregará antes de cada carácter especial un `'\'`, el cual le indica al intérprete de la base de datos que debe ignorar cualquier acción especial que tenga el siguiente carácter. Por ejemplo, el carácter especial `'\n'` generalmente se interpreta como un salto de línea, al cambiarlo por `'\\n'`, se ignora la acción de escribir el salto de línea y se muestra textualmente.

## Validar la entrada con una lista

Existen dos formas de realizar este método:

### Lista negra

La idea es comparar la entrada proporcionada por el usuario contra una lista de valores que no están permitidos. Para tener una lista negra<sup>4</sup> efectiva, hay que considerar cualquier posible entrada que pudiera ser maliciosa, y a medida que los hackers descubran nuevas técnicas de evadirla, actualizar la lista negra en consecuencia.

Algunas palabras que se pueden introducir en la lista son:

- |         |            |           |              |
|---------|------------|-----------|--------------|
| ▪ -     | ▪ varchar  | ▪ delete  | ▪ open       |
| ▪ ;     | ▪ nvarchar | ▪ drop    | ▪ select     |
| ▪ /*    | ▪ alter    | ▪ end     | ▪ sys        |
| ▪ */    | ▪ begin    | ▪ exec    | ▪ sysobjects |
| ▪ @@    | ▪ cast     | ▪ execute | ▪ syscolumns |
| ▪ @     | ▪ create   | ▪ fetch   | ▪ table      |
| ▪ char  | ▪ cursor   | ▪ insert  | ▪ update     |
| ▪ nchar | ▪ declare  | ▪ kill    |              |

Esta lista está compuesta, entre otros, de caracteres que delimitan comentarios en *SQL*, nombres de tipos de datos, y nombres de operaciones que se pueden realizar sobre los datos almacenados y el esquema interno de la base de datos.

<sup>4</sup>Ejemplo de implementación de una lista negra en *ASP*. <https://blogs.iis.net/nazim/filtering-sql-injection-from-classic-asp>

## Lista blanca

En contraparte, se puede crear una lista blanca, donde se puedan comparar las entradas con una lista de valores que se esperan para un campo determinado.

Por ejemplo, el uso de una expresión regular para determinar si los datos de entrada de un campo de correo electrónico tienen el formato correcto.

Aunque las listas blancas son más complicadas de implementar, son un método más efectivo para filtrar ataques ya que únicamente aceptan valores válidos para cierto campo. Por otro lado, las listas negras dependen su extensión para fungir como un filtro efectivo.

## Parametrización de variables

Este método consiste en pre compilar una consulta en *SQL* de modo que sólo se tengan que agregar los parámetros para poder ejecutarse. Las consultas pre compiladas, como la siguiente, funcionan como plantillas que el servidor podrá usar de manera repetitiva y eficiente con diferentes parámetros.

```
PREPARE prepStmt FROM 'SELECT * FROM Users WHERE id = ?';
SET @idNum = value;
EXECUTE prepStmt USING @idNum;
DEALLOCATE PREPARE prepStmt;
```

Como podemos ver en el ejemplo anterior, se crea una declaración de la consulta que contendrá la entrada del usuario. Después de esto se asigna un valor a los parámetros de la consulta. `@idNum = value`, lo que permite que la base de datos sea capaz de distinguir entre la consulta y la entrada, de modo que un hacker no pueda cambiar el resultado esperado de la consulta. Por ejemplo, si se le da la entrada `1` o `1=1`, en el formato correspondiente, la consulta pre compilada buscará un usuario cuyo `id` coincida con la entrada dada. Finalmente se realiza la consulta de manera habitual y se prepara para su siguiente uso.

Estas son sólo algunas técnicas para evitar los ataques de inyección de código *SQL*. Conforme los hackers encuentran formas de evadirlas se crean nuevas técnicas de prevención<sup>5</sup> con el fin de evitar que cumplan su objetivo malicioso.

## Conclusión

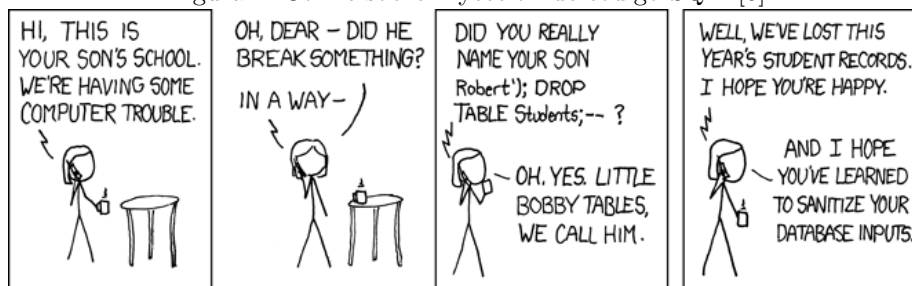
Los hackers constantemente están buscando vulnerabilidades en los servidores web. Cualquier campo de entrada podría ser una posible llave de acceso para vulnerar un sitio web, por lo que al implementar o actualizar algún módulo de

---

<sup>5</sup>Repositorio de la *OWASP* donde se en listan técnicas para prevenir ataques de inyección de código *SQL*. [https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/SQL\\_Injection\\_Prevention\\_Cheat\\_Sheet.md](https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.md)

un servidor hay que suponer que cualquier dato que se ingrese al sistema, independientemente de su origen, es malicioso hasta que se demuestre lo contrario. De igual modo, hay que tener presente que un ataque puede llegar de cualquier dirección, muchas empresas enfocan su atención en blindar los servidores de sus sitios web primarios (como el sistema de compras) dejando a sus sitios secundarios expuestos, cuando en ocasiones estos tienen acceso a los mismos datos que los sitios primarios o contienen datos igual de valiosos. Es fundamental que cualquier sistema esté protegido ante este tipo de ataques, ya que en caso de no hacerlo se compromete la veracidad de los datos que se almacenan y se expone la información privada de los usuarios.

Figura 2: Cómico sobre Inyección de código *SQL*. [6]



## Referencias

- [1] AO Kaspersky Lab. ¿qué es una inyección de sql y cómo evitarla? — kaspersky. URL <https://latam.kaspersky.com/resource-center/definitions/sql-injection>.
- [2] LulzSec. Sownage press release, Jun 2011. URL [https://web.archive.org/web/20110603061641/http://lulzsecurity.com/releases/sownage\\_PRETENTIOUS%20PRESS%20STATEMENT.txt](https://web.archive.org/web/20110603061641/http://lulzsecurity.com/releases/sownage_PRETENTIOUS%20PRESS%20STATEMENT.txt).
- [3] Open Web Application Security Project. Testing for sql injection (otg-inpval-005), Apr 2016. URL [https://www.owasp.org/index.php/Testing\\_for\\_SQL\\_Injection\\_\(OTG-INPVAL-005\)](https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-005)).
- [4] Paul Rubens. How to prevent sql injection attacks, May 2018. URL <https://www.esecurityplanet.com/threats/how-to-prevent-sql-injection-attacks.html>.
- [5] TeamSQL. Sql injection 101, May 2017. URL <https://teamsql.io/blog/?p=954>.
- [6] xkcd. Exploits of a mom. *A webcomic of romance, sarcasm, math, and language*. URL <https://xkcd.com/327/>.