4. Riesgos de seguridad en aplicaciones Web

SQL Injection

Víctor Ponz

Institut Educació Secundària
El Caminàs

Curso de especialización en Ciberseguridad

Contenidos

¿Qué es?

Según la Wikipedia

> **Inyección SQL** es un método de infiltración de código intruso que se vale de una vulnerabilidad informática presente en una aplicación en el nivel de validación de las entradas para realizar operaciones sobre una base de datos.
>
> El origen de la vulnerabilidad radica en la incorrecta comprobación o filtrado de las variables utilizadas en un programa que contiene, o bien genera, código SQL. Es, de hecho, un error de una clase más general de vulnerabilidades que puede ocurrir en cualquier lenguaje de programación o script que esté incrustado en otro.
>
> Se conoce como Inyección SQL, indistintamente, al tipo de vulnerabilidad, al método de infiltración, al hecho de incrustar código SQL intruso y a la porción de código incrustado.

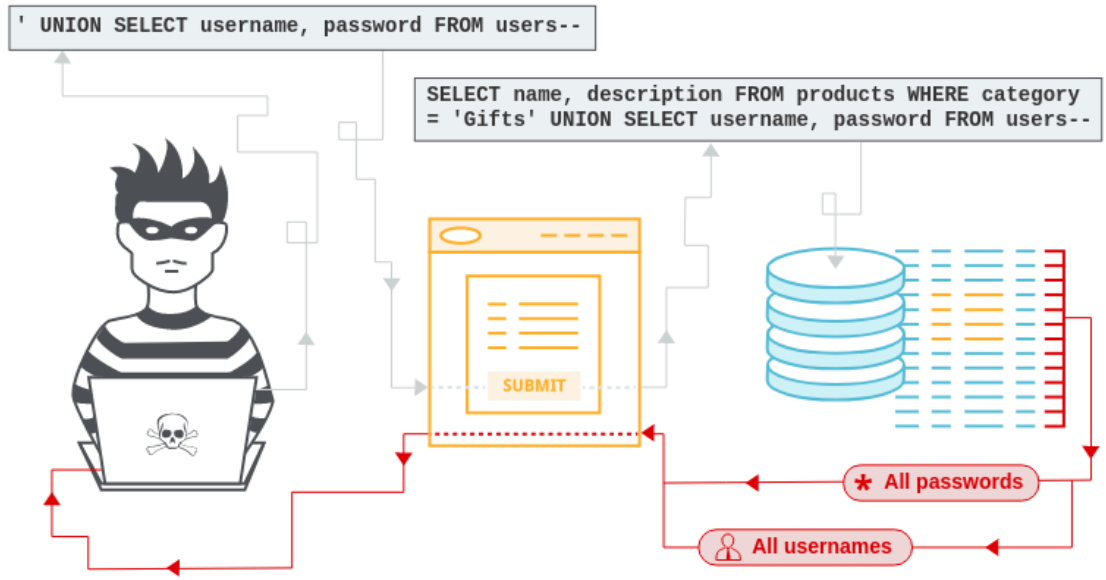

Figure 1: SQL Inyection

Obtener datos ocultos

Vamos a ver unas consultas básicas de una web de ejemplo que es insegura porque no ha sido programada correctamente. En concreto, la página de productos se puede hacker mediante SQL Injection (SQLi)

Para obtener el listado de los productos se accede a la URL

`https://insecure-website.com/products?category=Gifts`

Y la petición se transforma en la siguiente consulta:

```
SELECT * FROM produts
WHERE category = 'Gifts'
AND released = 1
```

El atacante de una web insegura puede lanzar esta petición

`https://insecure-website.com/products?category=Gifts'--`

que en el backend se convierte en la siguiente consulta:

```
SELECT * FROM produts
WHERE category = 'Gifts'--'
AND released = 1
```

Los dos guiones (–) se consideran un comentario, lo que acaba de ocurrir es que la base de datos va a ignorar `AND released = 1` provocando que se puedan obtener todos los productos de la Categoría Gifts sin tener en cuenta la bandera `released`

Lo que convierte la consulta anterior en esta:

```
SELECT * FROM produts
WHERE category = 'Gifts'
```

Si la página atacada funciona de esta forma, es muy sencillo hacer una consulta que devuelva todos los productos:

Petición

`https://insecure-website.com/products?category=Gifts'+OR+1=1--`

Se convierte en

```
SELECT * FROM produts
WHERE category = 'Gifts'
OR 1=1--' AND released = 1
```

Como 1=1 es siempre cierto, se obtienen TODOS los productos de la página

> **Realizad todos los laboratorios de esta actividad en un único documento.**
>
> Para cada uno de ellos debéis incluir una captura de pantalla como que lo habéis logrado. Por ejemplo, el siguiente muestra que lo he resuelto

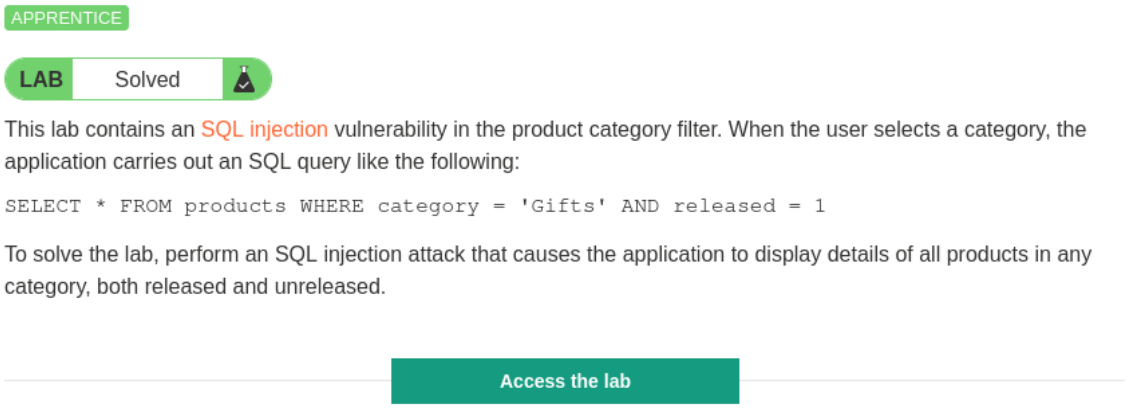

Figure 2: Laboratorio 1

> **LAB 1**
>
> Vulnerabilidad de inyección SQL en la cláusula WHERE que permite la recuperación de datos ocultos

Subvertir la funcionalidad de la página

Ahora consideramos una aplicación con un formulario de login típico

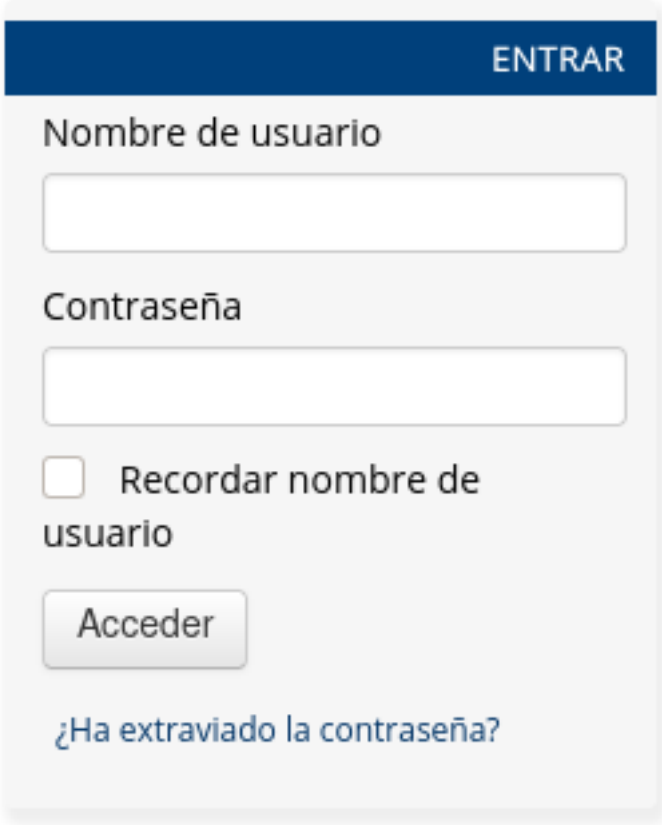

Figure 3: Formulario de login

Cuando un usuario introduce `Víctor` y password `hola` se genera la siguiente consulta:

```
SELECT * FROM users WHERE
username = 'Víctor' AND
password = 'hola'
```

En este escenario, un atacante puede entrar como cualquier otro usuario cuyo nombre conozca simplemente haciendo que la password esté comentada;

Por ejemplo, si introduce `administrator'--` entonces la aplicación genera la siguiente consulta

```
SELECT * FROM users WHERE
username = 'administrator' -- AND
password = ''
```

> **LAB 2** Vulnerabilidad de inyección SQL que permite eludir el inicio de sesión

Inyección SQL de segundo orden

Además, podemos lanzar más de una sentencia. Por ejemplo:

```
SELECT * FROM users WHERE
username = 'administrator'; UPDATE users SET password='letmein' where user =
 ↪  'administrator'-- AND
password = ''
```

Obtener datos de otras tablas

Se puede obtener mediante una consulta `UNION` que añada una nueva sentencia sql del tipo `SELECT * FROM table`

Si por ejemplo la aplicación tiene una consulta del tipo

```
SELECT name, description
FROM products WHERE
category = 'Gifts'
```

El atacante puede hacerle un ataque de `UNION`

```
SELECT name, description
FROM products WHERE
category = 'Gifts' UNION
SELECT username, password
FROM users--
```

Lo que permite obtener el listado de todos los usuarios y contraseñas del sistema.

Cuando una aplicación es vulnerable a la inyección de SQL y los resultados de la consulta se devuelven dentro de las respuestas de la aplicación, la palabra clave `UNION` se puede utilizar para recuperar datos de otras tablas dentro de la base de datos. Esto da como resultado un ataque `UNION` de inyección SQL.

La palabra clave `UNION` le permite ejecutar una o más consultas `SELECT` adicionales y agregar los resultados a la consulta original. Por ejemplo:

```
SELECT a, b FROM table1 UNION SELECT c, d FROM table2
```

Esta consulta SQL devolverá un único conjunto de resultados con dos columnas, que contienen los valores de las columnas **a** y **b** en la **tabla1** y las columnas **c** y **d** en la **tabla2**.

Para que una consulta UNION funcione, se deben cumplir dos requisitos clave:

1. Las consultas individuales deben devolver el mismo número de columnas.
2. Los tipos de datos de cada columna deben ser compatibles entre las consultas individuales.

Para llevar a cabo un ataque UNION de inyección SQL, debe asegurarse de que su ataque cumpla con estos dos requisitos. Esto generalmente implica averiguar:

1. ¿Cuántas columnas se devuelven de la consulta original?
2. ¿Qué columnas devueltas de la consulta original son de un tipo de datos adecuado para contener los resultados de la consulta inyectada?

Determinar el número de columnas necesarias en un ataque UNION de inyección SQL

Al realizar un ataque UNION de inyección SQL, existen dos métodos efectivos para determinar cuántas columnas se devuelven desde la consulta original.

El primer método implica inyectar una serie de cláusulas ORDER BY e incrementar el índice de columna especificado hasta que se produzca un error. Por ejemplo, suponiendo que el punto de inyección es una cadena entre comillas dentro de la cláusula UNION de la consulta original, enviaría:

```
' ORDER BY 1--
' ORDER BY 2--
' ORDER BY 3--
etc.
```

Esta serie de cargas útiles modifica la consulta original para ordenar los resultados por diferentes columnas en el conjunto de resultados. La columna en una cláusula ORDER BY se puede especificar por su índice, por lo que no necesita saber los nombres de ninguna columna. **Cuando el índice de columna** especificado excede el número de columnas reales en el conjunto de resultados, la base de datos **devuelve un error**, parecido a:

```
La posición ORDER BY número 3 está fuera del rango del número de elementos en la lista
```

La aplicación puede devolver el error de la base de datos en su respuesta HTTP, o puede devolver un error genérico o simplemente no devolver ningún resultado. Siempre que pueda detectar alguna diferencia en la respuesta de la aplicación, puede inferir cuántas columnas se devuelven desde la consulta.

El segundo método implica enviar una serie de cargas útiles de `UNION SELECT` que especifican un número diferente de valores nulos:

```
'UNION SELECT NULL--
'UNION SELECT NULL, NULL--
'UNION SELECT NULL, NULL, NULL--
etc.
```

Si el **número de nulos no coincide** con el número de columnas, la base de datos devuelve un error, como por ejemplo:

```
Todas las consultas combinadas con un operador UNION, INTERSECT o EXCEPT deben tener
```

Una vez más, la aplicación podría devolver este mensaje de error, o podría devolver un error genérico o ningún resultado. Cuando el número de nulos coincide con el número de columnas, la base de datos devuelve una fila adicional en el conjunto de resultados, que contiene valores nulos en cada columna. El efecto sobre la respuesta HTTP resultante depende del código de la aplicación. Si tiene suerte, verá contenido adicional dentro de la respuesta, como una fila adicional en una tabla HTML. De lo contrario, los valores nulos pueden desencadenar un error diferente, como un `NullPointerException`. En el peor de los casos, la respuesta puede ser indistinguible de la causada por un número incorrecto de nulos, lo que hace que este método para determinar el recuento de columnas sea ineficaz.

> **LAB 3** Ataque UNION de inyección SQL, determinando el número de columnas devueltas por la consulta

Examinar la base de datos

Siguiendo el patrón anterior, es posible acceder a la versión de SQL que está ejecutando el servidor mediante la consulta (en Oracle)

```
SELECT * FROM v$version
```

En la mayoría de sistemas gestores de bases de datos se pueden obtener todas las tablas con la siguiente consulta:

```
SELECT * FROM
information_schema.tables
```

Consultar el tipo y versión de del servidor de base de datos

Database type	Query
Microsoft, MySQL	SELECT @@version
Oracle	SELECT * FROM v$version
PostreSQL	SELECT version()

Por ejemplo, se puede usar un ataque de tipo UNION

```
' UNION SELECT @@version..
```

> **LAB 4**. Averigua la versión de la base de datos mediante un ataque de UNION

Blind SQL Injections

La inyección SQL ciega surge cuando una aplicación es vulnerable a la inyección SQL, pero sus respuestas HTTP no contienen los resultados de la consulta SQL relevante o los detalles de los errores de la base de datos.

Con las vulnerabilidades de inyección ciega de SQL, muchas técnicas, como los ataques UNION, no son efectivas porque se basan en poder ver los resultados de la consulta inyectada dentro de las respuestas de la aplicación. Todavía es posible explotar la inyección SQL ciega para acceder a datos no autorizados, pero se deben utilizar diferentes técnicas.

¿Qué es la inyección SQL ciega? La inyección SQL ciega surge cuando una aplicación es vulnerable a la inyección SQL, pero sus respuestas HTTP no contienen los resultados de la consulta SQL relevante o los detalles de los errores de la base de datos.

Con las vulnerabilidades de inyección ciega de SQL, muchas técnicas, como los ataques `UNION`, no son efectivas porque se basan en poder ver los resultados de la consulta inyectada dentro de las respuestas de la aplicación. Todavía es posible explotar la inyección SQL ciega para acceder a datos no autorizados, pero se deben utilizar diferentes técnicas. Explotación de la inyección SQL ciega mediante la activación de respuestas condicionales

Explotación de la inyección SQL ciega mediante la activación de respuestas condicionales Considere una aplicación que utiliza cookies de seguimiento para recopilar análisis sobre el uso. Las solicitudes a la aplicación incluyen un encabezado de cookie como este:

`Cookie: TrackingId = u5YD3PapBcR4lN3e7Tj4`

Cuando se procesa una solicitud que contiene una cookie `TrackingId`, la aplicación determina si se trata de un usuario conocido mediante una consulta SQL como esta:

```
SELECT TrackingId FROM TrackedUsers WHERE TrackingId =
 ↪  'u5YD3PapBcR4lN3e7Tj4'
```

Esta consulta es vulnerable a la inyección de SQL, pero los resultados de la consulta no se devuelven al usuario. Sin embargo, la aplicación se comporta de manera diferente dependiendo de si la consulta devuelve datos. Si devuelve datos (porque se envió un `TrackingId` reconocido), se muestra un mensaje de "Bienvenido de nuevo" dentro de la página.

Este comportamiento es suficiente para poder explotar la vulnerabilidad de inyección SQL ciega y recuperar información activando diferentes respuestas de forma condicional, dependiendo de una condición inyectada. Para ver cómo funciona esto, suponga que se envían dos solicitudes que contienen los siguientes valores de cookie `TrackingId` a su vez:

`… Xyz' AND 1 = 1… Xyz AND 1 = 2`

El primero de estos valores hará que la consulta devuelva resultados, porque la condición inyectada `AND 1 = 1` es verdadera, por lo que se mostrará el mensaje "Bienvenido de nuevo". Mientras que el segundo valor hará que la consulta no devuelva ningún resultado, porque la condición inyectada es falsa, por lo que no se mostrará el mensaje "Bienvenido de nuevo". Esto nos permite determinar la respuesta a cualquier condición inyectada individual y así extraer los datos de uno en uno.

Por ejemplo, suponga que hay una tabla llamada `Users` con las columnas `Username` y `Password`, y un usuario llamado Administrador. Podemos determinar sistemáticamente la contraseña para este usuario enviando una serie de entradas para probar la contraseña un carácter a la vez.

Para hacer esto, comenzamos con la siguiente entrada:

```
xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1,
```

Esto devuelve el mensaje "Bienvenido de nuevo", que indica que la condición inyectada es verdadera, por lo que el primer carácter de la contraseña es mayor que `m`.

A continuación, enviamos la siguiente entrada:

```
xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1,
```

Esto **no** devuelve el mensaje "Bienvenido de nuevo", lo que indica que la condición inyectada es falsa, por lo que el primer carácter de la contraseña no es mayor que `t`.

Finalmente, enviamos la siguiente entrada, que devuelve el mensaje "Bienvenido de nuevo", confirmando así que el primer carácter de la contraseña es `s`:

```
xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1,
```

Podemos continuar este proceso para determinar sistemáticamente la contraseña completa para el usuario Administrador.

> **LAB 5** Inyección SQL ciega con respuestas condicionales La haremos con Burp Suite más adelante

Como prevenir la inyección SQL

La mayoría de las instancias de inyección de SQL se pueden prevenir mediante el uso de consultas parametrizadas (también conocidas como declaraciones preparadas) en lugar de la concatenación de cadenas dentro de la consulta.

El siguiente código es vulnerable a la inyección de SQL porque la entrada del usuario se concatena directamente en la consulta:

```
 String query = "SELECT * FROM products WHERE category = '"+ input + "'";

Statement statement = connection.createStatement();

ResultSet resultSet = statement.executeQuery(query);
```

Este código se puede reescribir fácilmente de manera que evite que la entrada del usuario interfiera con la estructura de la consulta:

```
PreparedStatement statement = connection.prepareStatement("SELECT * FROM
 ↪  products WHERE category = ?");

statement.setString(1, input);

ResultSet resultSet = statement.executeQuery();
```

Las consultas parametrizadas se pueden usar para cualquier situación en la que la entrada que no es de confianza aparece como datos dentro de la consulta, incluida la cláusula `WHERE` y los valores en una instrucción `INSERT` o `UPDATE`. No se pueden usar para manejar entradas que no sean de confianza en otras partes de la consulta, como nombres de tablas o columnas, o la cláusula `ORDER BY`. La funcionalidad de la aplicación que coloca datos que no son de confianza en esas partes de la consulta deberá adoptar un enfoque diferente, como la inclusión de valores de entrada permitidos en la lista blanca o el uso de una lógica diferente para ofrecer el comportamiento requerido.

Para que una consulta parametrizada sea eficaz en la prevención de la inyección de SQL, la cadena que se utiliza en la consulta siempre debe ser una constante codificada de forma rígida y nunca debe contener datos variables de ningún origen. No se sienta tentado a decidir caso por caso si un elemento de datos es confiable y continúe usando la concatenación de cadenas dentro de la consulta para los casos que se consideran seguros. Es muy fácil cometer errores sobre el posible origen de los datos o que los cambios en otro código infrinjan las suposiciones sobre qué datos están contaminados.

Tenéis a vuestra disposición una hoja de trucos de SQL injection

Obtenido y traducido de

https://portswigger.net/web-security/sql-injection

Para saber más

https://cheatsheetseries.owasp.org/cheatsheets/Injection_Prevention_Cheat_Sheet.html

https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

https://owasp.org/www-community/attacks/SQL_Injection_Bypassing_WAF