Prevención de SQL Injection en el Modelo de Usuarios y Roles

¿Qué es un SQL Injection?

El SQL Injection es una técnica de ataque en la que un atacante inserta o "inyecta" código SQL malicioso en las consultas de la base de datos de una aplicación. Esto puede permitir al atacante manipular las consultas, recuperar información sensible, modificar datos, o incluso eliminar datos de la base de datos. Los SQL Injections son posibles cuando la entrada del usuario no es debidamente validada o filtrada, permitiendo que comandos SQL maliciosos se ejecuten en el sistema.

Este tipo de vulnerabilidad es una de las más comunes y peligrosas en aplicaciones web y sistemas que interactúan con bases de datos, como el modelo relacional de **Usuarios y Roles** descrito anteriormente.

¿Cómo se lleva a cabo un SQL Injection?

Supongamos que queremos obtener el perfil de un usuario basado en su nombre de usuario (`username`). Una consulta SQL típica en una aplicación mal protegida podría ser la siguiente:

```
SELECT * FROM user WHERE username = 'input_del_usuario';
```

Si el valor de `input_del_usuario` es una entrada controlada por el usuario y no se valida ni se protege correctamente, un atacante podría inyectar código SQL adicional, por ejemplo:

- Entrada legítima: `john`

  ```
  SELECT * FROM user WHERE username = 'john';
  ```

- Entrada maliciosa: `john' OR '1'='1`

  ```
  SELECT * FROM user WHERE username = 'john' OR '1'='1';
  ```

La consulta maliciosa devolvería todos los registros de la tabla `user` porque la condición `OR '1'='1'` siempre es verdadera.

Ejemplos de SQL Injection en el Modelo Relacional

Dado el modelo de base de datos que tenemos, aquí hay algunos ejemplos de cómo un atacante podría intentar explotar SQL Injection y cómo evitarlos.

Ejemplo 1: Consulta de Usuario por Nombre de Usuario

Supongamos que en el sistema hay una función para recuperar datos del usuario basándose en el nombre de usuario. Una consulta SQL podría verse así:

```
SELECT * FROM user WHERE username = 'input_del_usuario';
```

- **Entrada de ataque**: `admin' --`
  - **Consulta generada**:

    ```
    SELECT * FROM user WHERE username = 'admin' --';
    ```

    o

    ```
    SELECT * FROM user WHERE username = 'admin' --;
    ```

  - **Resultado**: Si `admin` es un nombre de usuario válido, la consulta devolverá los datos de ese usuario. El comentario (`--`) evita que el resto de la consulta se ejecute, permitiendo al atacante obtener información sin restricciones.

Ejemplo 2: Modificación de Roles

En el sistema, podríamos tener una consulta para asignar un rol a un usuario usando el ID del usuario y el ID del rol. Una consulta podría ser la siguiente:

```
UPDATE user_role SET role_id = 'input_role_id' WHERE user_id = 'input_user_id';
```

Si los valores `input_role_id` y `input_user_id` no se validan correctamente, un atacante podría realizar un ataque de SQL Injection:

- **Entrada de ataque**:
  - `input_user_id`: `1; DELETE FROM user WHERE '1'='1`
  - **Consulta generada**:

    ```
    UPDATE user_role SET role_id = 'input_role_id' WHERE user_id = 1;
    DELETE FROM user WHERE '1'='1';
    ```

  - **Resultado**: Esta consulta eliminaría todos los registros de la tabla `user`, ya que `WHERE '1'='1'` siempre es verdadero.

Prevención de SQL Injection

Existen varias prácticas y técnicas recomendadas para prevenir los ataques de SQL Injection. Aquí las más importantes aplicadas al modelo de **Usuarios y Roles**.

1. **Uso de Sentencias Preparadas (Prepared Statements)**

Las sentencias preparadas permiten separar los parámetros de la consulta, evitando que el código SQL inyectado se interprete como parte de la consulta. Esta técnica se utiliza para evitar que el contenido de las variables del usuario sea interpretado como código SQL.

Ejemplo en pseudocódigo para una consulta segura:

```
stmt = conn.prepare("SELECT * FROM user WHERE username = ?");
stmt.setString(1, input_del_usuario);
stmt.executeQuery();
```

Con esta técnica, incluso si `input_del_usuario` contiene código SQL malicioso, no se ejecutará porque la entrada se trata como un simple valor y no como parte de la consulta.

2. **Validación y Escape de Entrada**

Para reducir el riesgo de SQL Injection, es crucial validar y filtrar toda la entrada del usuario. Algunas recomendaciones son:

- **Validar los tipos de datos**: Por ejemplo, si `user_id` debe ser un número, asegurarse de que la entrada es un número.
- **Escapar caracteres especiales**: Caracteres como `'`, `"`, `--`, y `;` deben escaparse adecuadamente en caso de ser utilizados.

Ejemplo:

```
SELECT * FROM user WHERE username = 'input_del_usuario';
```

Aquí, si el valor de `input_del_usuario` es escapado, cualquier intento de incluir caracteres especiales se manejará de forma segura.

3. **Uso de ORM (Object-Relational Mapping)**

Las bibliotecas ORM (como Sequelize para Node.js, Hibernate para Java, etc.) ayudan a prevenir SQL Injection al abstraer la interacción con la base de datos. Los ORM generan las consultas automáticamente, utilizando mecanismos seguros para la entrada de los usuarios.

Ejemplo en un ORM:

```
User.findOne({ where: { username: input_del_usuario } });
```

Al utilizar el método `findOne` de un ORM, no se realiza una concatenación directa de la consulta, lo que ayuda a evitar SQL Injection.

4. **Políticas de Mínimo Privilegio**

Limitar los permisos de las cuentas de usuario que interactúan con la base de datos. La cuenta utilizada para la aplicación debe tener acceso solo a las operaciones necesarias y no a la eliminación o modificación de tablas enteras si no es necesario.

5. **Evitar el Uso de Consultas Dinámicas**

Evitar construir consultas dinámicas concatenando variables directamente en el SQL. En su lugar, se deben usar sentencias preparadas y parámetros de consulta.

Ejemplo inseguro:

```
SELECT * FROM user WHERE username = '" + username + "' AND password = '" +
password + "'";
```

Ejemplo seguro:

```
stmt = conn.prepare("SELECT * FROM user WHERE username = ? AND password = ?");
stmt.setString(1, username);
stmt.setString(2, password);
stmt.executeQuery();
```

Ejemplo de Seguridad Aplicada al Modelo Relacional

Supongamos que queremos implementar una consulta para asignar un rol a un usuario en el modelo relacional de **Usuarios y Roles**. La consulta segura usando sentencias preparadas podría verse así:

```
-- Consulta para asignar un rol a un usuario usando sentencias preparadas
stmt = conn.prepare("INSERT INTO user_role (user_id, role_id) VALUES (?, ?)");
stmt.setLong(1, user_id);
stmt.setLong(2, role_id);
stmt.executeUpdate();
```

Aquí, los valores de `user_id` y `role_id` se pasan como parámetros, evitando la ejecución de código malicioso.