UT03_03 Explicación de problemas de inyección de código SQL y PreparedStatement

1. Qué es la inyección de código. (Fuentes: Wikipedia y tutorialspoints).

Se dice que existe o se produjo una *inyección SQL* cuando, de alguna manera, se inserta o "inyecta" código SQL invasor dentro del código SQL programado, a fin de alterar el funcionamiento normal del programa y lograr así que se ejecute la porción de código "invasor" incrustado, en la base de datos.

Este tipo de intrusión normalmente es de carácter malicioso, dañino o espía, por tanto es un problema de seguridad informática, y debe ser tomado en cuenta por el programador de la aplicación para poder prevenirlo. Un programa elaborado con descuido, displicencia o con ignorancia del problema, podrá resultar ser vulnerable, y la seguridad del sistema (base de datos) podrá quedar eventualmente comprometida.

La intrusión ocurre durante la ejecución del programa vulnerable, ya sea, en computadores de escritorio o bien en sitios Web, en este último caso obviamente ejecutándose en el servidor que los aloja.

La vulnerabilidad se puede producir automáticamente cuando un programa "arma descuidadamente" una sentencia SQL en tiempo de ejecución, o bien durante la fase de desarrollo, cuando el programador explicita la sentencia SQL a ejecutar en forma desprotegida. En cualquier caso, siempre que el programador necesite y haga uso de parámetros a ingresar por parte del usuario, a efectos de consultar una base de datos; ya que, justamente, dentro de los parámetros es donde se puede incorporar el código SQL intruso.

Al ejecutarse la consulta en la base de datos, el código SQL inyectado también se ejecutará y podría hacer un sinnúmero de cosas, como insertar registros, modificar o eliminar datos, autorizar accesos e, incluso, ejecutar otro tipo de código malicioso en el computador.

Por ejemplo, asumiendo que el siguiente código reside en una aplicación web y que existe un parámetro "nombreUsuario" que contiene el nombre de usuario a consultar, una inyección SQL se podría provocar de la siguiente forma:

El código SQL original y *vulnerable* es:

```
consulta:= "SELECT * FROM usuarios WHERE nombre = '" + nombreUsuario + "';"
```

Si el operador escribe un nombre, por ejemplo "Alicia", nada anormal sucederá, la aplicación generaría una sentencia SQL similar a la siguiente, que es perfectamente correcta, en donde se seleccionarían todos los registros con el nombre "Alicia" en la base de datos:

```
SELECT * FROM usuarios WHERE nombre = 'Alicia';
```

Pero si un operador malintencionado escribe como nombre de usuario a consultar:

```
Alicia'; DROP TABLE usuarios; SELECT * FROM datos WHERE nombre LIKE '%
```

, se generaría la siguiente consulta SQL, (el color verde es lo que pretende el programador, el azul es el dato, y el rojo, el código SQL inyectado):

```
SELECT * FROM usuarios WHERE nombre = 'Alicia';
DROP TABLE usuarios;
SELECT * FROM datos WHERE nombre LIKE '%';
```

En la base de datos se ejecutaría la consulta en el orden dado, se seleccionarían todos los registros con el nombre 'Alicia', se borraría la tabla 'usuarios' y finalmente se seleccionaría toda la tabla "datos", que no debería estar disponible para los usuarios web comunes.

En resumen, cualquier dato de la base de datos puede quedar disponible para ser leído o modificado por un usuario malintencionado.

Nótese por qué se llama "**Inyección**" SQL. Si se observa el código malicioso, de color rojo, se notará que está insertado en el medio del código bueno, el verde. Así, el código rojo ha sido "**inyectado**" dentro del verde.

La inyección SQL es fácil de evitar, por parte del programador, en la mayoría de los lenguajes de programación que permiten desarrollar aplicaciones web. En la siguiente sección se trata brevemente ese tema.

1.1 Blind SQL injection

'*Ataque a ciegas por **inyección SQL'***, en inglés, **Blind SQL injection**, es una técnica de ataque que utiliza la inyección SQL. Su evidencia cuando en una página web, por una falla de seguridad, no se muestran mensajes de error al no producirse resultados correctos ante una consulta a la base de datos, mostrándose siempre el mismo contenido (es decir, solo hay respuesta si el resultado es correcto).

Sentencias condicionales con el tipo "Or 1=1" o "having 1=1" ofrecen respuestas *siempre correctas* (true o verdadero) por lo cual suelen ser usadas por los programadores como formas de comprobación. El problema para la seguridad de la página radica en que esta técnica es utilizada en combinación con diccionarios o fuerza bruta para la búsqueda, carácter por carácter, de una contraseña, un nombre de usuario, un número de teléfono o cualquier otra información que albergue la base de datos atacada; para ello se utiliza código SQL específico que "va probando" cada carácter consiguiendo un resultado positivo acumulable cuando hay una coincidencia. De esta manera se puede saber, por ejemplo, que una contraseña comienza por "F...", luego continúa con ".i...", y luego "..r...", etc (acumula *Fir...*), hasta dar con la palabra completa.

Existen programas que automatizan este proceso de "tanteos" letra por letra en el resultado de la consulta SQL, que un intruso podría enviar inyectado.

1.2 Cómo evitarlo en JAVA

En lenguaje Java, se puede usar la clase `PreparedStatement`

En lugar de:

```
Connection con = (acquire Connection)
Statement stmt = con.createStatement();
```

```
 ResultSet rset = stmt.executeQuery("SELECT * FROM usuarios WHERE nombre = '"
+ nombreUsuario + "';");
```

se puede usar parametrización o escape de variables, como se indica en los siguientes apartados.

1.2.1 Parametrización de sentencias SQL

```
 Connection con = (acquire Connection)
 PreparedStatement pstmt = con.prepareStatement("SELECT * FROM usuarios WHERE
nombre = ?");
 pstmt.setString(1, nombreUsuario);
 ResultSet rset = pstmt.executeQuery();
```

1.2.2 Escape de las variables a insertar en la sentencia SQL

Escapar el texto contenido en la variable remplazando los caracteres especiales en SQL por su equivalente textual, de tal forma que SQL interprete todo el contenido de la variable como si fuera texto.

```
 Connection con = (acquire Connection)
 Statement stmt = con.createStatement();
 ResultSet rset = stmt.executeQuery("SELECT * FROM usuarios WHERE nombre = '"
+ nombreUsuario.replace("\\", "\\\\").replace("'", "\\'") + "';");
```

También se puede utilizar el método escapeSQL de la clase procedente de la librería de Apache Commons Lang

```
 Connection con = (acquire Connection)
 Statement stmt = con.createStatement();
 ResultSet rset = stmt.executeQuery("SELECT * FROM usuarios WHERE Nombre = '"
+ StringEscapeUtils.escapeSQL(NombreUsuario) + "';");
```

2. Prepared Statement o Consultas Preparadas

Como hemos visto el uso Prepared Statement nos ayuda a evitar la inyección de código Sql. Pero además cabe destacar que nos permite **pasar parámetros a las sentencias Sql**, y tienen un **mejor rendimiento, ya que son sentencias precompiladas y reutilizables.**

2.1 Funcionamiento

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   pstmt.close();
}
```

3. Ejemplo

```
/*
  * To change this license header, choose License Headers in Project
Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package preparedstatement;
   //STEP 1. Import required packages
import java.sql.*;

/**
 *
 * @author inmav
 */
public class Ejemplo_PS {
   // JDBC driver name and database URL
   static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
   static final String DB_URL = "jdbc:mysql://localhost/prueba";

   //  Database credentials
   static final String USER = "2daw";
   static final String PASS = "2daw";

   public static void main(String[] args) {
   Connection conn = null;
   PreparedStatement stmt = null;
   try{
      //STEP 2: Register JDBC driver
      Class.forName("com.mysql.jdbc.Driver");

      //STEP 3: Open a connection
      System.out.println("Connecting to database...");
      conn = DriverManager.getConnection(DB_URL,USER,PASS);

      //STEP 4: Execute a query
      System.out.println("Creating statement...");
      String sql = "UPDATE contacto set telefono=? WHERE nombre=?";
      stmt = conn.prepareStatement(sql);

      //Bind values into the parameters.
      stmt.setString(1, "444444444");  // This would set age
      stmt.setString(2, "Antonio"); // This would set ID

      // Let us update age of the record with ID = 102;
      int rows = stmt.executeUpdate();
      System.out.println("Rows impacted : " + rows );

      // Let us select all the records and display them.
      sql = "SELECT nombre, apellidos, telefono FROM contacto";
      ResultSet rs = stmt.executeQuery(sql);
```

```
      //STEP 5: Extract data from result set
      while(rs.next()){
         //Retrieve by column name
         String nombre  = rs.getString("nombre");
         String apellido = rs.getString("apellidos");
         String telefono = rs.getString("telefono");

         //Display values
         System.out.print("Nombre: " + nombre);
         System.out.print(", Apellido: " + apellido);
         System.out.println(", Teléfono: " + telefono);
      }
      //STEP 6: Clean-up environment
      rs.close();
      stmt.close();
      conn.close();
   }catch(SQLException se){
      //Handle errors for JDBC
      se.printStackTrace();
   }catch(Exception e){
      //Handle errors for Class.forName
      e.printStackTrace();
   }finally{
      //finally block used to close resources
      try{
         if(stmt!=null)
            stmt.close();
      }catch(SQLException se2){
      }// nothing we can do
      try{
         if(conn!=null)
            conn.close();
      }catch(SQLException se){
         se.printStackTrace();
      }//end finally try
   }//end try
   System.out.println("Goodbye!");
}//end main
}//end JDBCExample
```