Reviewing Code for SQL Injection

From OWASP

«««« Main (Table of Contents) »»»»

Overview

A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system, and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands.

Related Security Activities

Description of SQL Injection Vulnerabilities

See the OWASP article on SQL Injection Vulnerabilities.

See the OWASP article on Blind_SQL_Injection Vulnerabilities.

How to Avoid SQL Injection Vulnerabilities

See the OWASP Development Guide article on how to Avoid SQL Injection Vulnerabilities.

How to Test for SQL Injection Vulnerabilities

See the OWASP Testing Guide article on how to Test for SQL Injection Vulnerabilities.

How to Locate Potentially Vulnerable Code

A secure way to build SQL statements is to construct all queries with PreparedStatement instead of Statement and/or to use parameterized stored procedures. Parameterized stored procedures are compiled before user input is added, making it impossible for a hacker to modify the actual SQL statement.

The account used to make the database connection must have “Least privilege.” If the application only requires read access then the account must be given read access only.

Avoid disclosing error information: Weak error handling is a great way for an attacker to profile SQL injection attacks. Uncaught SQL errors normally give too much information to the user and contain things like table names and procedure names.

Best Practices when Dealing with Databases

Use Database stored procedures, but even stored procedures can be vulnerable. Use parameterized queries instead of dynamic SQL statements. Data validate all external input: Ensure that all SQL statements recognize user inputs as variables, and that statements are precompiled before the actual inputs are substituted for the variables in Java.

SQL Injection Example:

```
String DRIVER = "com.ora.jdbc.Driver";

String DataURL = "jdbc:db://localhost:5112/users";

String LOGIN = "admin";

String PASSWORD = "admin123";

Class.forName(DRIVER);

//Make connection to DB
Connection connection = DriverManager.getConnection(DataURL, LOGIN, PASSWORD);

String Username = request.getParameter("USER"); // From HTTP request

String Password = request.getParameter("PASSWORD"); // From HTTP request

int iUserID = -1;

String sLoggedUser = "";

String sel = "SELECT User_id, Username FROM USERS WHERE Username = '" +Username + "' AND Password = '" + Pass
```

```
Statement selectStatement = connection.createStatement ();
ResultSet resultSet = selectStatement.executeQuery(sel);


if (resultSet.next()) {

       iUserID = resultSet.getInt(1);
       sLoggedUser = resultSet.getString(2);
}

PrintWriter writer = response.getWriter ();

if (iUserID >= 0) {
       writer.println ("User logged in: " + sLoggedUser);
} else {

       writer.println ("Access Denied!")
}
```

When SQL statements are dynamically created as software executes, there is an opportunity for a security breach as the input data can truncate or malform or even expand the original SQL query!

Firstly, the request.getParameter retrieves the data for the SQL query directly from the HTTP request without any data validation (Min/Max length, Permitted characters, Malicious characters). This error gives rise to the ability to input SQL as the payload and alter the functionality in the statement.

The application places the payload directly into the statement causing the SQL vulnerability:

String sel = "SELECT User_id, Username FROM USERS WHERE Username = '" Username + "' AND Password = '" + Password + "'";

.NET

Parameter collections such as SqlParameterCollection provide type checking and length validation. If you use a parameters collection, input is treated as a literal value, and SQL Server does not treat it as executable code, and therefore the payload cannot be injected. Using a parameters collection lets you enforce type and length checks. Values outside of the range trigger an exception. Make sure you handle the exception correctly. Example of the SqlParameterCollection:

```
using System.Data;

using System.Data.SqlClient;

using (SqlConnection conn = new SqlConnection(connectionString))
{
  DataSet dataObj = new DataSet();

  SqlDataAdapter sqlAdapter = new SqlDataAdapter( "StoredProc", conn);

  sqlAdapter.SelectCommand.CommandType = CommandType.StoredProcedure;
```

```
 //specify param type

  sqlAdapter.SelectCommand.Parameters.Add("@usrId", SqlDbType.VarChar, 15);

  sqlAdapter.SelectCommand.Parameters["@usrId "].Value = UID.Text; // Add data from user

  sqlAdapter.Fill(dataObj); // populate and execute proc

}
```

Stored procedures don’t always protect against SQL injection:

```
CREATE PROCEDURE dbo.RunAnyQuery
@parameter NVARCHAR(50)

AS
        EXEC sp_executesql @parameter
GO
```

The above procedure shall execute any SQL you pass to it. The directive sp_executesql is a system stored procedure in Microsoft® SQL Server™

Lets pass it.

```
DROP TABLE ORDERS;
```

Guess what happens? So we must be careful of not falling into the “We’re secure, we are using stored procedures” trap!

Classic ASP

For this technology you can use parameterized queries to avoid SQL injection attacks. Here is a good example:

```
<%
    option explicit
    dim conn, cmd, recordset, iTableIdValue

    'Create Connection
    set conn=server.createObject("ADODB.Connection")
    conn.open "DNS=LOCAL"

    'Create Command
    set cmd = server.createobject("ADODB.Command")
    With cmd
            .activeconnection=conn
            .commandtext="Select * from DataTable where Id = @Parameter"
            'Create the parameter and set its value to 1
            .Parameters.Append .CreateParameter("@Parameter", adInteger, adParamInput, , 1)
    End With
    'Get the information in a RecordSet
    set recordset = server.createobject("ADODB.Recordset")
    recordset.Open cmd, conn
    '....
    'Do whatever is needed with the information
    '....
    'Do clean up
    recordset.Close
    conn.Close
    set recordset = nothing
    set cmd = nothing
    set conn = nothing
%>
```

Notice that this is SQL Server Specific code. If you would use a ODBC/Jet connection to another DB which ISAM supports parameterized queries ,you should change your query to the following:

```
cmd.commandtext="Select * from DataTable where Id = ?"
```

Finally there is always a way of doing things **wrong** you can **(but should not)** do the following:

```
cmd.commandtext="Select * from DataTable where Id = " & Request.QueryString("Parameter")
```

«««« Main (Table of Contents) »»»»