SQL Injection

Impact of SQL injection - 1

- Bypass authentication or even impersonate specific users.
- Complete disclosure of data residing on a database server
- Alter data stored in a database affecting data integrity (financial transactions)
- Delete data from a database affecting an application’s availability until the database is restored (if backup exists)
- Executing system commands and use this as an initial vector in an attack of an internal network that sits behind a firewall

Impact of SQL injection - 2

- Confidentiality: Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL Injection vulnerabilities.
- Authentication: If poor SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password.
- Authorization: If authorization information is held in a SQL database, it may be possible to change this information through the successful exploitation of a SQL Injection vulnerability.
- Integrity: Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with a SQL Injection attack.

Performing SQL injection

- Two conditions are required:
  - a relational database that uses SQL
  - and a user controllable input which is directly used in an SQL query

Defending against SQL injection

- Prepared Statements (with Parameterized Queries)
  - are simple to write
  - easier to understand than dynamic queries
  - force the developer to
    - first define all the SQL code
    - then pass in each parameter to the query later
  - allows the database to distinguish between code and data, regardless of what user input is supplied
  - ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker
  - input validation or proper escaping
  - Java sample:

```
String custname = request.getParameter("customerName"); // This should REALLY be validated too
 // perform input validation to detect attacks
 String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";

 PreparedStatement pstmt = connection.prepareStatement( query );
 pstmt.setString( 1, custname);
 ResultSet results = pstmt.executeQuery( );
```

Defending against SQL injection - 2

- Stored Procedures
  - They require the developer to just build SQL statements with parameters
  - The SQL code for a stored procedure is defined and stored in the database itself
  - Use input validation or proper escaping
  - VB.NET sample:

```
Try
    Dim command As SqlCommand = new SqlCommand("sp_getAccountBalance", connection)
    command.CommandType = CommandType.StoredProcedure
    command.Parameters.Add(new SqlParameter("@CustomerName", CustomerName.Text))
    Dim reader As SqlDataReader = command.ExecuteReader()
    ‘ …
Catch se As SqlException
    ‘ error handling
End Try
```

Defending against SQL injection – 3

- Escaping All User Supplied Input
  - escape user input before putting it in a query
  - cannot guarantee it will prevent all SQL Injection in all situations
  - Use DBMS related character escaping schemes specific to certain kinds of queries
  - DBMS will not confuse that input with SQL code
  - Bad example:

```
String query = "SELECT user_id FROM user_data WHERE user_name = '" + req.getParameter("userID")
 + "' and user_password = '" + req.getParameter("pwd") +"'";
 try {
    Statement statement = connection.createStatement( … );
    ResultSet results = statement.executeQuery( query );
 }
```

  - Better example:

```
Encoder oe = new OracleEncoder();
 String query = "SELECT user_id FROM user_data WHERE user_name = '"
  + oe.encode( req.getParameter("userID")) + "' and user_password = '"
  + oe.encode( req.getParameter("pwd")) +"'";
```

Defending against SQL injection – 4

- Least Privilege
  - Minimize the privileges assigned to every database account
  - Do not assign DBA or admin type access rights to your application accounts
  - Determine what access rights your application accounts require
  - Minimize the privileges of the operating system account that the DBMS runs under
  - Different DB users could be used for different web applications

SQL Injection examples

- SQL Injection Based on 1=1 is Always True

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId; => UserId == “105 or 1=1”
```

- SQL Injection Based on ""="" is Always True

```
uName = getRequestString("UserName");          => UserName == " or ""="
uPass = getRequestString("UserPass");          => UserPass == " or ""="
sql = "SELECT * FROM Users WHERE Name ='" + uName + "' AND Pass ='" + uPass + "'“
```

- SQL Injection Based on Batched SQL Statements

```
txtUserId = getRequestString("UserId");        => UserId == “105; DROP TABLE Suppliers”
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```