



# SQL Injection Attack

# Brief Tutorial of SQL

- MySQL: an open-source relational database management system
- **Log in to MySQL**

```
$ mysql -uroot -pseedubuntu
Welcome to the MySQL monitor.
...
mysql>
```

- **Create a Database:**
  - “SHOW DATABASES” command: list existing databases.
  - Create a new database called dbtest

```
mysql> SHOW DATABASES;
.....
mysql> CREATE DATABASE dbtest;
```

# SQL Tutorial: Create a Table

- A relational database organizes its data using tables.
- Create a table “employee” with seven attributes (i.e., columns) for the database “dbtest”

○ “USE dbtest” - let the system know which database to use as there may be multiple databases

○ “DESCRIBE” to display the structure of the table

```
mysql> USE dbtest
mysql> CREATE TABLE employee (
  ID      INT (6) NOT NULL AUTO_INCREMENT,
  Name    VARCHAR (30) NOT NULL,
  EID     VARCHAR (7) NOT NULL,
  Password VARCHAR (60),
  Salary  INT (10),
  SSN     VARCHAR (11),
  PRIMARY KEY (ID)
);
mysql> DESCRIBE employee;
```

Field	Type	Null	Key	Default	Extra
ID	int(6)	NO	PRI	NULL	auto_increment
Name	varchar(30)	NO		NULL	
EID	varchar(30)	NO		NULL	
Password	varchar(60)	YES		NULL	
Salary	int(10)	YES		NULL	
SSN	varchar(11)	YES		NULL	

# SQL Tutorial: Insert a Row

- INSERT INTO - to insert a new record into a table

```
mysql> INSERT INTO employee (Name, EID, Password, Salary, SSN)
      VALUES ('Ryan Smith', 'EID5000', 'paswd123', 80000,
      '555-55-5555');
```

- Insert a record into the “employee” table.
- We do not specify a value of the ID column (Primary Key) as it is automatically set by the database.

# SQL Tutorial: SELECT Statement

SELECT - retrieves information from a database

- the most common operation

```
mysql> SELECT * FROM employee;
```

ID	Name	EID	Password	Salary	SSN
1	Alice	EID5000	passwd123	80000	555-55-5555
2	Bob	EID5001	passwd123	80000	555-66-5555
3	Charlie	EID5002	passwd123	80000	555-77-5555
4	David	EID5003	passwd123	80000	555-88-5555

Asks the database for all its records, including all the columns

```
mysql> SELECT Name, EID, Salary FROM employee;
```

Name	EID	Salary
Alice	EID5000	80000
Bob	EID5001	80000
Charlie	EID5002	80000
David	EID5003	80000

Asks the database only for Name, EID and Salary columns

# SQL Tutorial: WHERE Clause

- Uncommon for a SQL query to retrieve all records in a database.
- WHERE clause - set conditions for SQL statements (e.g., SELECT, UPDATE, DELETE)

```
mysql> SQL Statement  
      WHERE predicate;
```

The above SQL statement only reflects the rows for which the predicate in the WHERE clause is TRUE.

- Predicate
  - a logical expression
  - combined using keywords AND and OR

# SQL Tutorial: WHERE Clause

- First query - a record with EID5001 in EID field
- Second query - records with EID='EID5001' or Name='David'

```
mysql> SELECT * FROM employee WHERE EID='EID5001';
+----+-----+-----+-----+-----+-----+
| ID | Name | EID      | Password | Salary | SSN          |
+----+-----+-----+-----+-----+-----+
|  2 | Bob  | EID5001 | paswd123 | 80000 | 555-66-5555 |
+----+-----+-----+-----+-----+-----+

mysql> SELECT * FROM employee WHERE EID='EID5001' OR Name='David';
+----+-----+-----+-----+-----+-----+
| ID | Name  | EID      | Password | Salary | SSN          |
+----+-----+-----+-----+-----+-----+
|  2 | Bob   | EID5001 | paswd123 | 80000 | 555-66-5555 |
|  4 | David | EID5003 | paswd123 | 80000 | 555-88-5555 |
+----+-----+-----+-----+-----+-----+
```

# SQL Tutorial: WHERE Clause

- If the condition is always True, then all the rows are affected by the SQL statement

```
mysql> SELECT * FROM employee WHERE 1=1;
```

ID	Name	EID	Password	Salary	SSN
1	Alice	EID5000	passwd123	80000	555-55-5555
2	Bob	EID5001	passwd123	80000	555-66-5555
3	Charlie	EID5002	passwd123	80000	555-77-5555
4	David	EID5003	passwd123	80000	555-88-5555

- This 1=1 predicate looks quite useless in real queries, but it will become useful in SQL Injection attacks



# SQL Tutorial: UPDATE Statement

- UPDATE - to modify an existing record

```
mysql> UPDATE employee SET Salary=82000 WHERE Name='Bob';
mysql> SELECT * FROM employee WHERE Name='Bob';
+----+-----+-----+-----+-----+-----+
| ID | Name | EID      | Password | Salary | SSN          |
+----+-----+-----+-----+-----+-----+
|  2 | Bob  | EID5001 | paswd123 | 82000  | 555-66-5555 |
+----+-----+-----+-----+-----+-----+
```

# SQL Tutorial: Comments

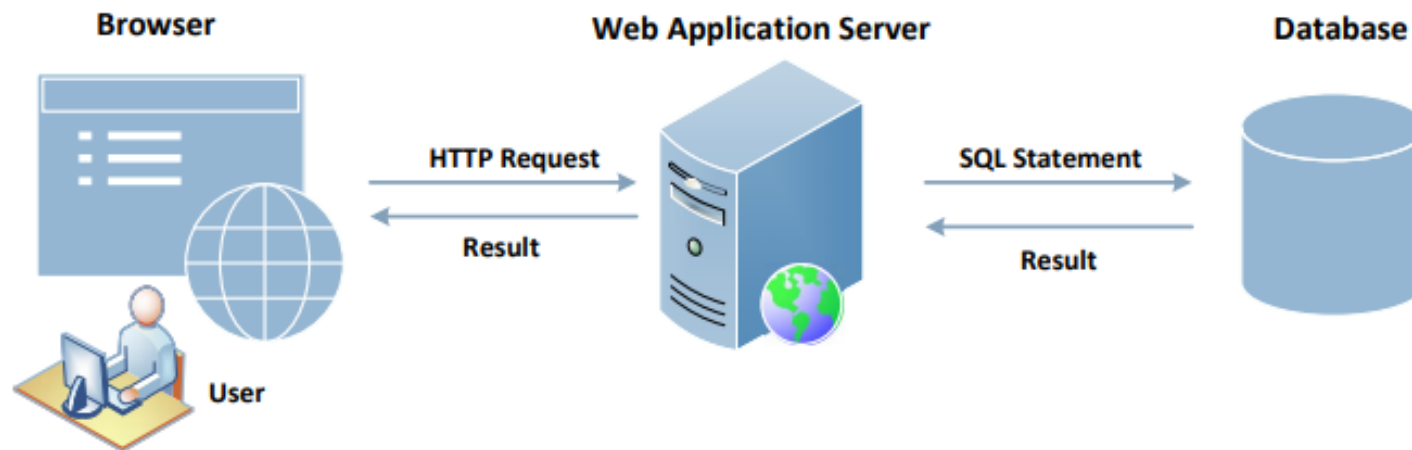
MySQL supports three comment styles

- Text from the # character to the end of line is treated as a comment
- Text from the "--" to the end of line is treated as a comment.
- Similar to C language, text between /\* and \*/ is treated as a comment

```
mysql> SELECT * FROM employee;    # Comment to the end of line
mysql> SELECT * FROM employee;    -- Comment to the end of line
mysql> SELECT * FROM /* In-line comment */ employee;
```

# Interacting with Database in Web Application

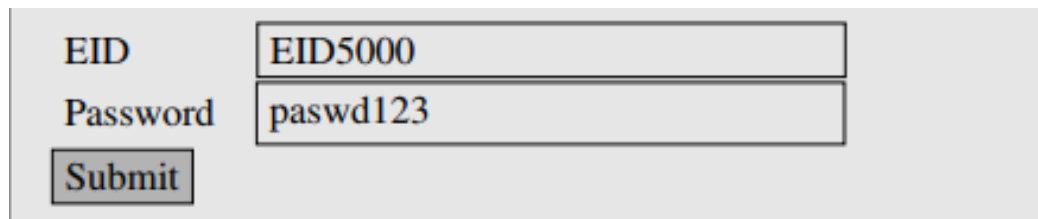
- A typical web application consists of three major components:



- SQL Injection attacks can cause damage to the database.
  - Users do not directly interact with the database, but through a web server.
  - If this channel is not securely implemented, malicious users can attack the database.

# Getting Data from User

- Example: a form where users can type their data.  
Once the submit button is clicked, send out an HTTP request with the data attached



A screenshot of a web form. It contains two text input fields. The first field is labeled "EID" and contains the text "EID5000". The second field is labeled "Password" and contains the text "paswd123". Below these fields is a button labeled "Submit".

- HTML source of the above form

```
<form action="getdata.php" method="get">  
  EID:      <input type="text" name="EID"><br>  
  Password: <input type="text" name="Password"><br>  
           <input type="submit" value="Submit">  
</form>
```

# Getting Data from User

- HTTP GET requests - parameters are attached after the question mark in the URL
- Each parameter has a “name=value” pair and are separated by “&”
- For HTTPS, similar format, but encrypted data

```
http://www.example.com/getdata.php?EID=EID5000&Password=paswd123
```

- Once this request reached the target PHP script, the parameters inside the HTTP request will be saved to an array `$_GET` or `$_POST`.
- Example: a PHP script retrieves data from a GET request

```
<?php
    $eid = $_GET['EID'];
    $pwd = $_GET['Password'];
    echo "EID: $eid --- Password: $pwd\n";
?>
```

# How Web Applications Interact with Database

## Connecting to MySQL Database

- PHP program connects to the database server before conducting query on database using.
- E.g., `mysqli(...)` along with 4 arguments to create the database connection

```
function getDB() {
    $dbhost="localhost";
    $dbuser="root";
    $dbpass="seedubuntu";
    $dbname="dbtest";

    // Create a DB connection
    $conn = new mysqli($dbhost, $dbuser, $dbpass, $dbname);
    if ($conn->connect_error) {
        die("Connection failed: " . $conn->connect_error . "\n");
    }
    return $conn;
}
```

# How Web Applications Interact with Database

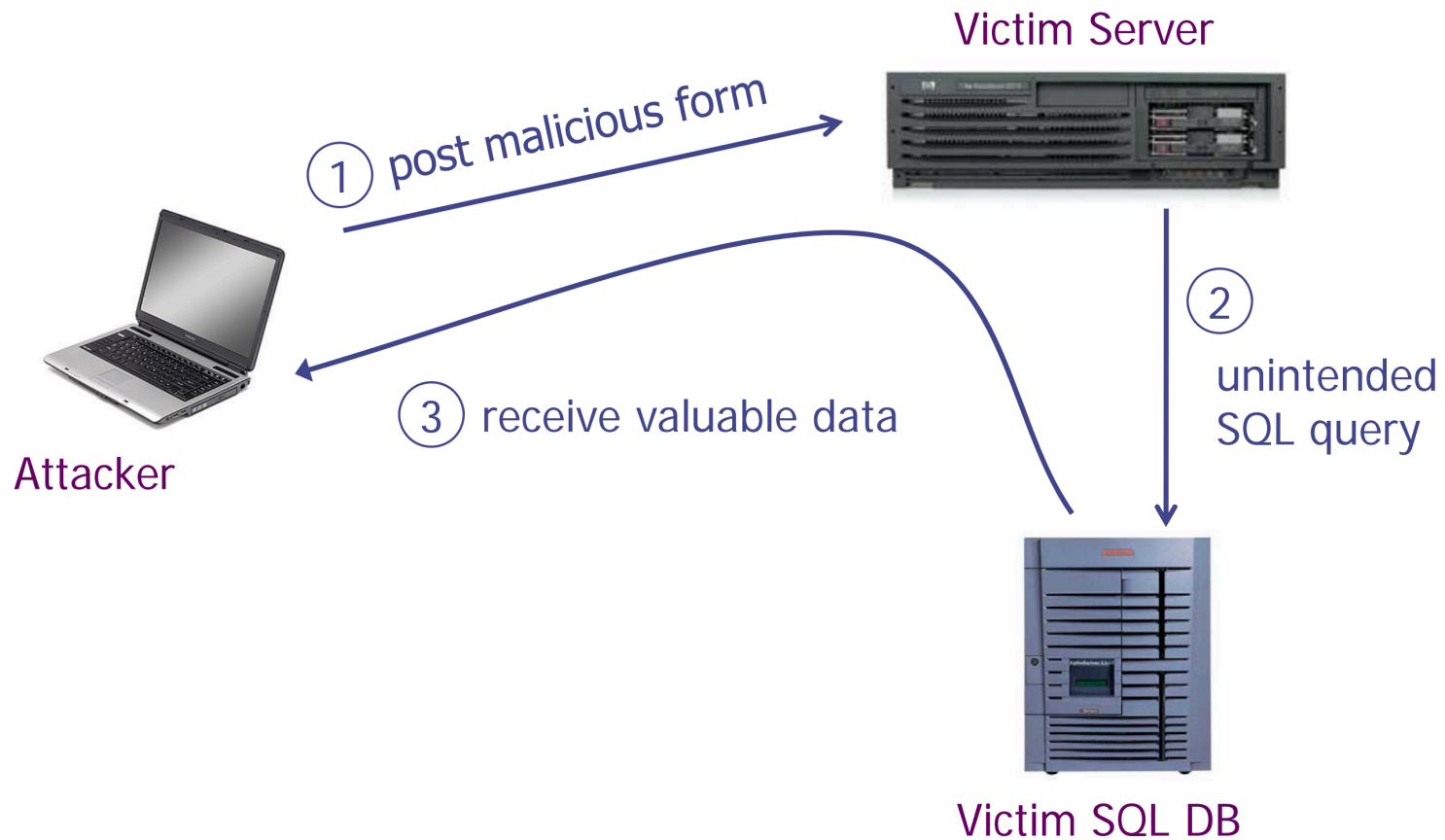
- Construct the query string and then send it to the database for execution.
- The channel between user and database creates a new attack surface for the database.

```
/* getdata.php */
<?php
    $eid = $_GET['EID'];
    $pwd = $_GET['Password'];

    $conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");
    $sql = "SELECT Name, Salary, SSN
           FROM employee
           WHERE eid= '$eid' and password='$pwd'";
    }      Constructing
           SQL statement

    $result = $conn->query($sql);
    if ($result) {
        // Print out the result
        while ($row = $result->fetch_assoc()) {
            printf ("Name: %s -- Salary: %s -- SSN: %s\n",
                   $row["Name"], $row["Salary"], $row['SSN']);
        }
        $result->free();
    }
    $conn->close();
?>
```

# SQL Injection Attacks





# CardSystems Attack

## CardSystems

- credit card payment processing company
- SQL injection attack in June 2005
- put out of business

## The Attack

- 263,000 credit card #s stolen from database
- credit card #s stored unencrypted
- 43 million credit card #s exposed



# Launching SQL Injection Attacks

- Everything provided by user will become part of the SQL statement.  
Is it possible for a user to change the meaning of the SQL statement?
- The intention of the blank fields below is for the user to provide input data.

```
SELECT Name, Salary, SSN
FROM employee
WHERE eid='  ' and password='  '
```

- Malicious user inputs
  - eid = EID5002' #
  - password = a random string

```
SELECT Name, Salary, SSN
FROM employee
WHERE eid= 'EID5002' #' and password='xyz'
```

# Launching SQL Injection Attacks

- Everything from the # sign to the end of line is considered as comment. The SQL statement is now equivalent to

```
SELECT Name, Salary, SSN  
FROM employee  
WHERE eid= 'EID5002'
```

- Now return the name, salary and SSN of the employee whose EID is EID5002 even though the user doesn't know the employee's password.
- How to get all the records from the database without knowing all the EID's?
- Create a predicate for WHERE clause so that it is true for all records.

```
SELECT Name, Salary, SSN  
FROM employee  
WHERE eid= 'a' OR 1=1
```

# Launching SQL Injection Attacks using cURL

- Using a command-line tool to launch attacks.
  - Easier to automate attacks without a graphic user interface.
- Using cURL, we can send out a form from a command-line instead of from a web page.

```
% curl 'www.example.com/getdata.php?EID=a' OR 1=1 #&Password='
```

- Will the above command work?
- No. In an HTTP request, special characters in the attached data need to be encoded or they may be mis-interpreted.
- In the above URL, we need to encode the apostrophe, whitespace and the # sign

```
% curl 'www.example.com/getdata.php?EID=a%27%20
                                     OR%201=1%20%23&Password='
Name: Alice -- Salary: 80000 -- SSN: 555-55-5555<br>
Name: Bob -- Salary: 82000 -- SSN: 555-66-5555<br>
Name: Charlie -- Salary: 80000 -- SSN: 555-77-5555<br>
Name: David -- Salary: 80000 -- SSN: 555-88-5555<br>
```

# Modify Database

- If the statement is UPDATE or INSERT INTO, we will have chance to change the database.
- Consider the form created for changing passwords. It asks users to fill in three pieces of information, EID, old password and new password.
- When Submit button is clicked, an HTTP POST request will be sent to the server-side script `changepasswd.php`, which uses an UPDATE statement to change the user's password.

EID	<input type="text" value="EID5000"/>
Old Password	<input type="text" value="paswd123"/>
New Password	<input type="text" value="paswd456"/>
<input type="submit" value="Submit"/>	

```
/* changepasswd.php */
<?php
    $eid = $_POST['EID'];
    $oldpwd = $_POST['OldPassword'];
    $newpwd = $_POST['NewPassword'];

    $conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");
    $sql = "UPDATE employee
            SET password='$newpwd'
            WHERE eid= '$eid' and password='$oldpwd'";

    $result = $conn->query($sql);
    $conn->close();
?>
```

# Modify Database

- Assume that Alice (EID5000) is not satisfied with her salary. She would like to increase her own salary using the SQL injection vulnerability. How?

New Password

- By typing the above string in “New Password” box, we get the UPDATE statement to set one more attribute for us, the salary attribute. The SQL statement will now look as follows.

```
UPDATE employee
SET password='paswd456', salary=100000 #'
WHERE eid= 'EID5000' and password='paswd123' ";
```

- What if Alice does not like Bob and would like to reduce Bob’s salary to 0, but she only knows Bob’s EID (eid5001), not his password. How can she execute the attack?

EID	<input type="text" value="EID5001' #"/>
Old Password	<input type="text" value="anything"/>
New Password	<input type="text" value="paswd456', salary=0 #"/>

# Multiple SQL Statements

- Damages are bounded because we cannot change everything in the existing SQL statement.
- It will be more dangerous if we can cause the database to execute an arbitrary SQL statement.
- To append a new SQL statement “DROP DATABASE dbtest” to the existing SQL statement to delete the entire dbtest database, we can type the following in the EID box

```
EID  a'; DROP DATABASE dbtest; #
```

- The resulting SQL statement is equivalent to the following, where we have successfully appended a new SQL statement to the existing SQL statement string.

```
SELECT Name, Salary, SSN  
FROM employee  
WHERE eid= 'a'; DROP DATABASE dbtest;
```

- The above attack doesn't work against MySQL, because in PHP's mysqli extension, the mysqli::query() API doesn't allow multiple queries to run in the database server.

# Multiple SQL Statements

- Execute two SQL statements using the `$mysqli->query()` API

```
/* testmulti_sql.php */
<?php
$mysqli = new mysqli("localhost", "root", "seedubuntu", "dbtest");
$res     = $mysqli->query("SELECT 1; DROP DATABASE dbtest");
if (!$res) {
    echo "Error executing query: (" .
        $mysqli->errno . ") " . $mysqli->error;
}
?>
```

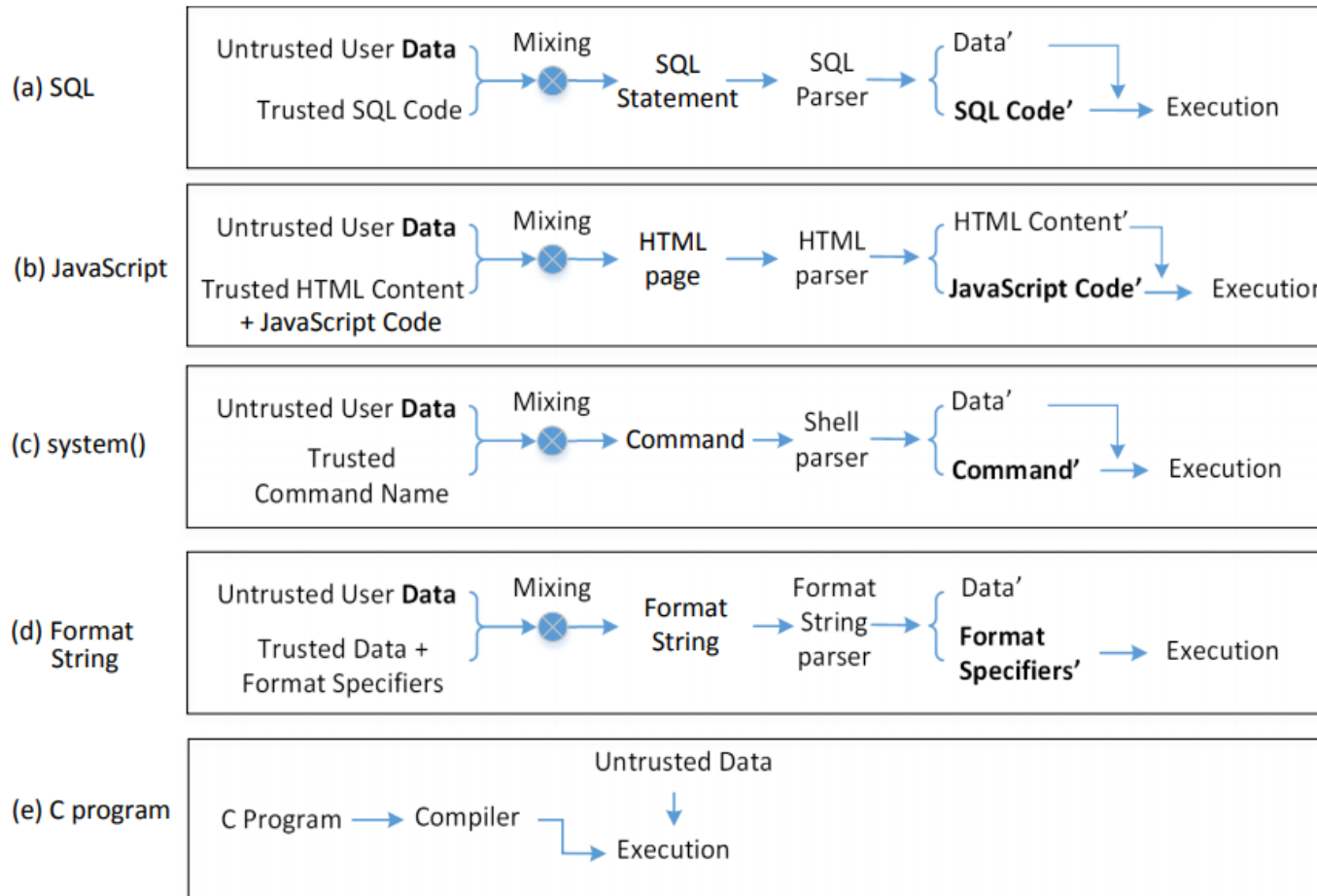
- Error message after executing the code

```
$ php testmulti_sql.php
Error executing query: (1064) You have an error in your SQL syntax;
check the manual that corresponds to your MySQL server version
for the right syntax to use near 'DROP DATABASE dbtest' at line 1
```

- To run multiple SQL statements, one can use `$mysqli->multi_query()` [not recommended]



# The Fundamental Cause



## Mixing data and code

- SQL Injection attacks
- XSS attacks
- Attacks on the system() function
- Format string attacks

# Countermeasures: Filtering and Encoding Data

- Before mixing user-provided data with code, inspect the data. Filter out any character that may be interpreted as code.
- Special characters are commonly used in SQL Injection attacks. To get rid of them, encode them.
- Encoding a special character tells parser to treat the encoded character as data and not as code.

```
Before encoding:  aaa' OR 1=1 #  
After encoding:  aaa\' OR 1=1 #
```

- PHP's mysqli extension has a built-in method called `mysqli::real_escape_string()`. It can be used to encode the characters that have special meanings in SQL. The following code snippet shows how to use this API.

```
/* getdata_encoding.php */  
<?php  
    $conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");  
    $eid = $mysqli->real_escape_string($_GET['EID']);           ①  
    $pwd = $mysqli->real_escape_string($_GET['Password']);     ②  
    $sql = "SELECT Name, Salary, SSN  
            FROM employee  
            WHERE eid= '$eid' and password=' $pwd' ";  
?>
```

# Countermeasures: Prepared Statement

- **Fundament cause** of SQL injection: mixing data and code
- **Fundament solution**: separate data and code.
- **Main Idea**: Sending code and data in separate channels to the database server. This way the database server knows not to retrieve any code from the data channel.
- **How**: using **prepared statement**
- An optimized feature that provides improved performance if the same or similar SQL statement needs to be executed repeatedly.
- Using prepared statements
  - Send an SQL statement template to the database, with certain values called parameters left unspecified
  - The database parses, compiles and performs query optimization on the SQL statement template and stores the result without executing it.
  - Later bind data to the prepared statement

# Countermeasures: Prepared Statement

```
$conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");  
$sql = "SELECT Name, Salary, SSN  
      FROM employee  
      WHERE eid= '$eid' and password='$pwd'";  
$result = $conn->query($sql);
```

← The vulnerable version: code and data are mixed together.

Using prepared statements, we separate code and data.

```
$conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");  
$sql = "SELECT Name, Salary, SSN  
      FROM employee  
      WHERE eid= ? and password=?";  
  
if ($stmt = $conn->prepare($sql)) {  
    $stmt->bind_param("ss", $eid, $pwd);  
    $stmt->execute();  
  
    $stmt->bind_result($name, $salary, $ssn);  
    while ($stmt->fetch()) {  
        printf ("%s %s %s\n", $name, $salary, $ssn);  
    }  
}
```

①

②

③

④

⑤

⑥

Send code

Send data

Start execution

## Why Are Prepared Statements Secure?

- Trusted code is sent via a code channel.
- Untrusted user-provided data is sent via data channel.
- Database clearly knows the boundary between code and data.
- Data received from the data channel is not parsed.
- Attacker may hide code in data, but the code will never be treated as code and thus not be executed.

# Summary

- Brief tutorial of SQL
- SQL Injection attacks and how to launch this type of attacks
- The fundament cause of the vulnerability
- Ways to defend against SQL Injection attacks
  - Prepared Statement