Web Security

Lecture 9

Instructor: C. Pu (Ph.D., Assistant Professor)

puc@marshall.edu

Introduction

- in real-world web app., data are stored in database
  - web app.: save data to or get data from a database
    - construct SQL statement
    - send SQL statement to the database
      - execute SQL statement
      - return the results back to web app.
- SQL statement usually contains user-provide data
  - what if a SQL statement is not constructed properly?
    - inject code into SQL statement
      - cause database to execute the code
      - SQL injection vulnerability

A 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>
```

login name password

mysql prompt: indicating login successfully

Note:
- no space between –u and login name
- no space between –p and password

- 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;
```

create database command

SQL commands are not case sensitive
- using upper-case to separate from non-commands in lower-case

MARSHALL

SQL Tutorial: Create a Table

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

select database to use

```
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    |                |
+----------+-------------+------+-----+---------+----------------+
```

define the structure of table 'employee'
- table columns are defined inside parentheses
  - each column contains
    - name, followed by type
    - number: maximum length
    - constraints (i.e., NOT NULL)

display the structure of table 'employee'

MARSHALL

SQL Tutorial: Insert a Row

- 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: Insert a Row

- the '*SELECT*' statement is the most common operation on databases
  - retrieves information from a database

all records

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

```
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 only for Name, EID and Salary columns

MARSHALL

- 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.

  - the above SQL statement only affects the rows for which the predicate in the 'WHERE' clause is TRUE
    - row for which predicate evaluates to FALSE or Unknown are not affected
  - the predicate is a logical expression
    - multiple predicates can be combined using keywords AND and OR

SQL Tutorial: WHERE Clause

```
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 |
+----+-------+---------+----------+--------+-------------+
```

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

MARSHALL

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
  - useful in SQL Injection attacks

MARSHALL

SQL Tutorial: UPDATE Statement

- use the UPDATE Statement to modify an existing record

multiple columns separated by comma

```
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 |
+----+------+---------+----------+--------+-------------+
```

MARSHALL

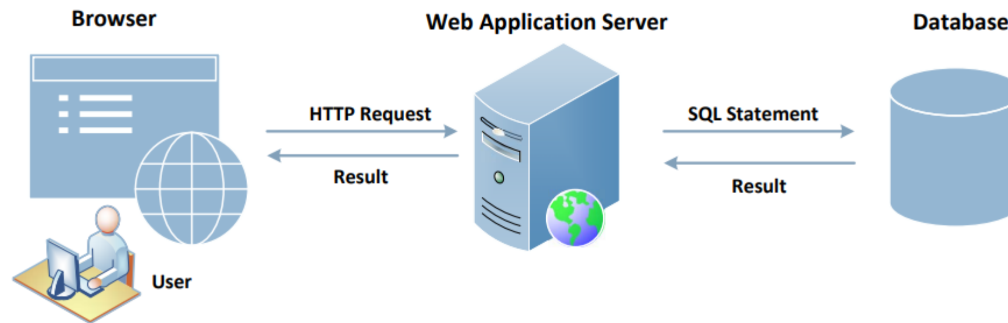
SQL Tutorial: Comments

- 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 (space)
    - this style requires the second dash to be followed by at least one whitespace or control character
  - similar to C language, text between /* and */ is treated as a comment
    - this style allows comment to be inserted into the middle of SQL statement; commend can span multiple lines

```
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:

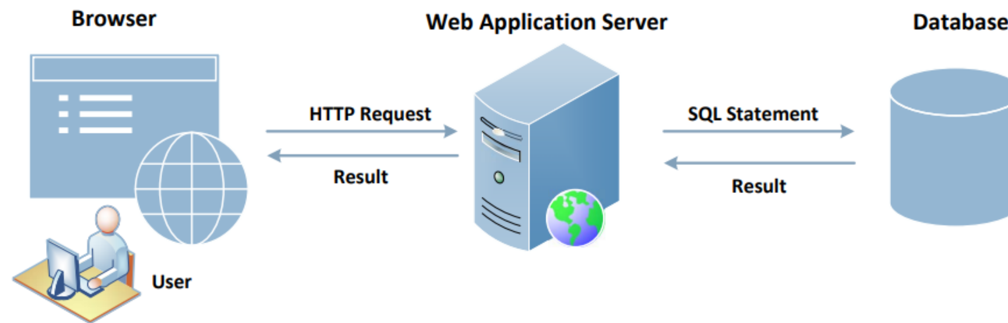


- web browser
  - get content; present content; interact with user; get user input
  - communicate with web app. server using HTTP
- web app. server
  - generate and deliver content to browser; rely on independent database server for data management
  - interact with database using SQL
- database

MARSHALL

Interacting with Database in Web Application

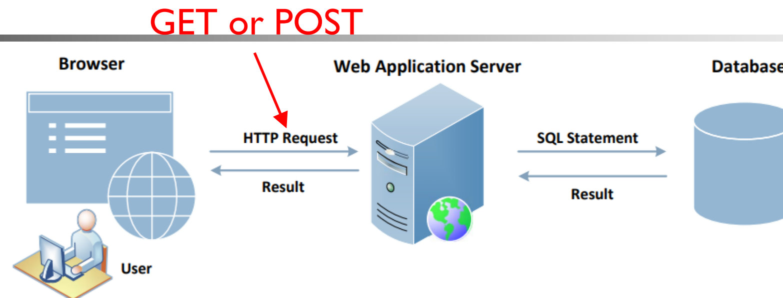
- a typical web application consists of three major components:



- SQL Injection attacks can cause damage to the database
- users do not directly interact with the database but through a web server
  - web app. server provide a channel for user's data to reach database
  - if this channel is not implemented properly, malicious users can attack the database

MARSHALL

Getting Data from User



- 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
Submit

- 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:

name of input field

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

MARSHALL

Getting Data from User

```
<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>
```

- HTTP GET request
  - the method field in the HTML code specified the GET type
  - in GET requests, parameters are attached after the question mark ? in the URL

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

  - 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 (getdata.php)
  - the parameters inside the HTTP request will be saved to an array $_GET or $_POST.
  - an example shows a PHP script getting data from a GET request

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

$_GET: an associative array of variables passed to the current script via the URL parameters

MARSHALL

How Web Applications Interact with Database

- once a user provides his/her EID and password to the serve-side script getdata.php
  - the script sends user's data (Name, salary, and SSN, along with correct password) back
- user data are actually stored in database
  - getdata.php needs to send a SQL query to database to get data

- three methods for PHP programs to interact with MySQL
  - PHP's MySQL Extension
  - PHP's MySQLi Extension
    - a relational database driver used in the PHP scripting language to provide an interface with MySQL databases
  - PHP Data Objects
    - defines a lightweight, consistent interface for accessing databases in PHP

How Web Applications Interact with Database

- connecting to MySQL Database
  - PHP program connects to the database server before conducting query on database using.
  - the code shown below uses new mysqli(…) along with its 4 arguments to create the database connection.

```
function getDB() {
    $dbhost="localhost";      ← host name
    $dbuser="root";           ← login name
    $dbpass="seedubuntu";     ← password
    $dbname="dbtest";         ← database name

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

MARSHALL

How Web Applications Interact with Database

- constructing a SQL query to fetch user's data
  - construct the query string
  - use mysqli::query() to 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'";

   $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();
?>
```

Constructing SQL statement

performs a query on the database

fetch the next row of a result set as an associative array

frees the memory associated with a result

MARSHALL

Launching SQL Injection Attacks

- user input will become part of the SQL statement
  - is it possible for a user to change the meaning of the SQL statement?
- example: the intention of the web app developer by the following is for user to provide some data for the blank areas

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

  - what if 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'
```

everything from # 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'
```

  - return the name, salary and SSN of the employee whose EID is EID5002 even though the user doesn't know the employee's password.
- let's see if a user can get all the records from the database
  - assuming that we don't know all the EID's in the database
  - 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
```

always true

- if the statement is UPDATE or INSERT INTO, we will have chance to change the database
- consider the form created for changing passwords.
  - 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
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 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 #

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 or encode them.
      - encoding a special character tells parser to treat the encoded character as data and not as code.
      - example

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

- PHP's mysqli extension has a built-in method

  mysqli::real_escape_string()

  - encode the characters that have special meanings in SQL.

```
/* 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'";
?>
```

prepends backslashes to the special characters

MARSHALL

Countermeasures: Prepared Statement

- best way to prevent SQL injection attack: separate code from data
  - data can never become code
- for SQL statement: sending code and data in separate channels to database server
  - database parser knows not to retrieve any code from the data channel
- SQL prepared statement
  - optimization feature: provides an improved performance if the same SQL statement needs to be executed repeatedly
    - send SQL statement template to the database with certain values left unspecified
      - database parses, compiles, and stores the result without executing it
      - at later time, we bind values to parameters, and ask database to execute

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: 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);
   }
}
```

send code (②)

send data (③)

start execution (④)

MARSHALL