

Lec 24: Database Security

CSED415: Computer Security
Spring 2024

Seulbae Kim

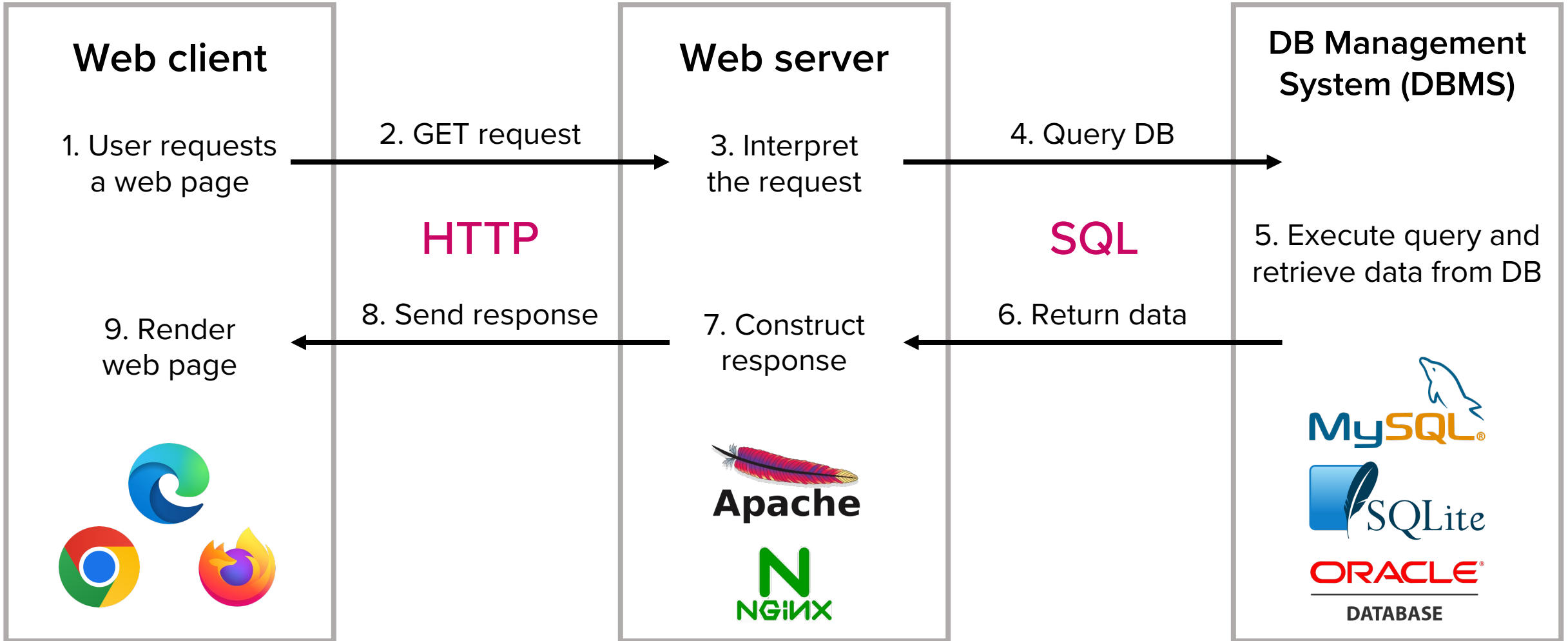
POSTECH
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Motivation

- Most websites need to store and retrieve data
 - User credentials, board posts and comments, product prices, ...
- The HTTP server is not designed to store and manage a large amount of data
 - HTTP server only handles HTTP requests
- Additional layer is required for data management

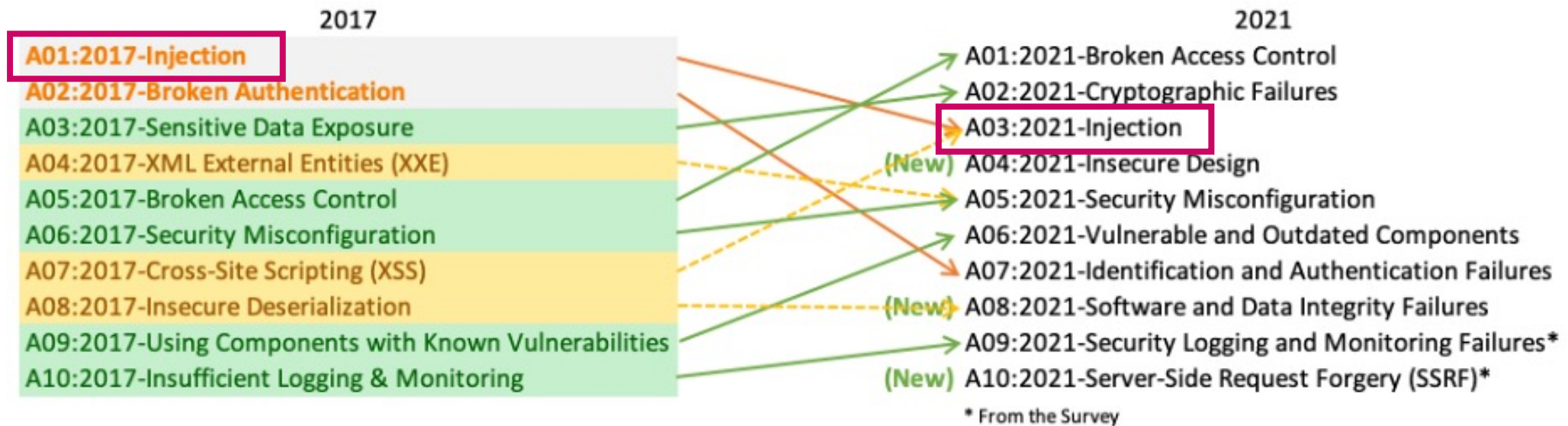
Most web applications rely on “database” systems

Structure of web services



OWASP Top 10 web security risks

- A standard awareness document that represents a broad consensus about the most critical security risks to web apps
 - <https://owasp.org/www-project-top-ten/>



Injection has been one of the top security risks!

Background: Database and SQL

Databases

- A database is a collection of tables
- A table is a collection of records
 - Table columns: Attributes
 - Table rows: Individual records

Table: students			
id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33

Structured Query Language (SQL)

- SQL is a language for database queries
 - Query: A request for retrieval or modification of data
 - DBMS executes a query and returns the result

SQL – SELECT

- Syntax: `SELECT [columns] FROM [table]`
 - Select columns from a table

Table: students			
id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33

SQL – SELECT

- Syntax: SELECT [columns] FROM [table]

```
SELECT name,age FROM students;
```

name	age
Alice	23
Bob	27
Claire	33

id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33

SQL – SELECT

- Syntax: SELECT [columns] FROM [table]

```
SELECT * FROM students;
```

id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33

id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33

SQL – WHERE

- Syntax: `SELECT [columns] FROM [table] WHERE [condition]`
 - Filter out selected rows
 - Arithmetic operators, comparison operators, and boolean operator are applicable

id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33

SQL – WHERE

- Syntax: SELECT [columns] FROM [table] WHERE [condition]

```
SELECT * FROM students  
WHERE field='Cybersecurity';
```

id	name	field	age
1	Alice	Cybersecurity	23

id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33

SQL – WHERE

- Syntax: SELECT [columns] FROM [table] WHERE [condition]

```
SELECT * FROM students  
WHERE age > 30;
```

id	name	field	age
3	Claire	Architecture	33

id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33

SQL – WHERE

- Syntax: SELECT [columns] FROM [table] WHERE [condition]

```
SELECT * FROM students  
WHERE age > 30 or id = 1;
```

id	name	field	age
1	Alice	Cybersecurity	23
3	Claire	Architecture	33

Table: students

id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33

SQL – ORDER BY


- Syntax: `SELECT [columns] FROM [table] ORDER BY [column]`
 - Sort the result in ascending (default) or descending order
 - Can use column name or column number

Table: students			
id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33

SQL – ORDER BY

- Syntax: SELECT [columns] FROM [table] ORDER BY [column]

```
SELECT * FROM students  
ORDER BY age DESC;
```



id	name	field	age
3	Claire	Architecture	33
2	Bob	Database	27
1	Alice	Cybersecurity	23

id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33

SQL – ORDER BY

- Syntax: SELECT [columns] FROM [table] ORDER BY [column]

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

Third column

id	name	field	age
3	Claire	Architecture	33
1	Alice	Cybersecurity	23
2	Bob	Database	27

id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33

SQL – INSERT INTO

- Syntax: INSERT INTO [table] VALUES [values]
 - Add rows into a table
 - VALUES specify the values for columns

Table: students			
id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33

SQL – INSERT INTO

- Syntax: INSERT INTO [table] VALUES [values]

```
INSERT INTO students  
VALUES (4, 'Dave', 'OS', 25);
```

Table: students

id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33
4	Dave	OS	25

SQL – INSERT INTO

- Syntax: INSERT INTO [table] VALUES [values]

```
INSERT INTO students  
VALUES (5, 'Eve', 'ML', 35);
```

id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33
4	Dave	OS	25
5	Eve	ML	35

SQL – UPDATE

- Syntax: UPDATE [table] SET [columns] = [values] WHERE [condition]
 - Change the values of existing rows in a table

Table: students			
id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33
4	Dave	OS	25
5	Eve	ML	35

SQL – UPDATE

- Syntax: UPDATE [table] SET [columns] = [values] WHERE [condition]

```
UPDATE students  
SET field = 'Graphics'  
WHERE id = 4;
```

Table: students

id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33
4	Dave	OS Graphics	25
5	Eve	ML	35

SQL – DELETE

- Syntax: DELETE FROM [table] WHERE [condition]
 - Delete rows of a certain condition from table

id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33
4	Dave	Graphics	25
5	Eve	ML	35

SQL – DELETE

- Syntax: DELETE FROM [table] WHERE [condition]

```
DELETE FROM students  
WHERE age > 30;
```

Table: students			
id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
3	Claire	Architecture	33
4	Dave	Graphics	25
5	Eve	ML	35

SQL – CREATE

- Syntax: CREATE TABLE [table] [columns]
 - Create a table with given columns

Table: students			
id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
4	Dave	Graphics	25

SQL – CREATE

- Syntax: CREATE TABLE [table] [columns]

```
CREATE TABLE professors (  
    id INT,  
    name VARCHAR(64),  
    field VARCHAR(64),  
    age INT  
);
```

Table: students			
id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
4	Dave	Graphics	25

Table: professors			
id	name	field	age

SQL – DROP

- Syntax: DROP TABLE [table]
 - Remove a table and all associated contents

Table: students			
id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
4	Dave	Graphics	25

Table: professors			
id	name	field	age

SQL – DROP

- Syntax: DROP TABLE [table]

```
DROP TABLE students
```

Table: students			
id	name	field	age
1	Alice	Cybersecurity	23
2	Bob	Database	27
4	Dave	Graphics	25

Table: professors			
id	name	field	age

SQL – Miscellaneous

- Comments: `--` (double dashes)
 - Works similarly to `//` in C or `#` in Python

```
SELECT name, age FROM stdents -- WHERE age < 30;
```

→ name and age of all records in the students table are selected

- Query delimiter: `;` (semicolon)

```
UPDATE professors SET age = 55 WHERE id = 4;  
SELECT age FROM professors where id = 4;
```

→ if row with id 4 exists, the row will be returned (age 55)

→ otherwise, empty result will be returned

SQL – Miscellaneous

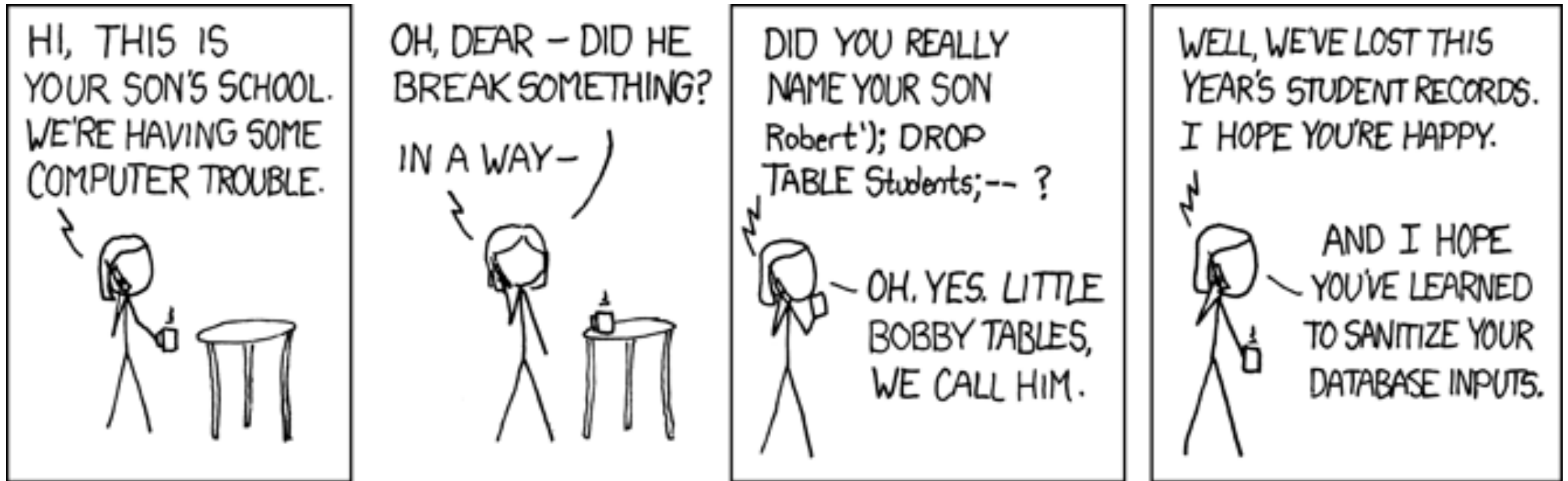
- Any many more..
 - You only need to know the basics for this class

SQL Injection Attack

SQL injection (SQLi)

- SQLi: Injecting malicious SQL queries into server to cause undesirable behavior of DBMS
 - Typically caused by vulnerable string manipulation logic for constructing SQL queries
 - Allows the attacker to execute arbitrary SQL queries on DBMS
 - Leak data
 - Add and/or modify records
 - Remove records and/or tables
 - Anything that can be done through SQL queries

“Exploits of a Mom”



<https://xkcd.com/327/>

SQLi – Anatomy of typical attacks

- A simple client-server interaction



name:

pass:

Submit



```
$query = "SELECT * FROM users WHERE name='$_GET["name"]' AND pass='$_GET["pass"]'";  
$result = mysqli_query($conn, $query);
```

Database queries are dynamically constructed from user inputs



Table: users

id	name	pass	age
1	Alice	1q2w3e4r	23
2	Bob	pa\$\$w0rd	27

SQLi – Anatomy of typical attacks

- A simple client-server interaction



name:

pass:



```
$query = "SELECT * FROM users WHERE name=' Alice ' AND pass=' 1q2w3e4r '";  
$result = mysqli_query($conn, $query);
```



Table: users			
id	name	pass	age
1	Alice	1q2w3e4r	23
2	Bob	pa\$\$w0rd	27

→ returned!

SQLi – Anatomy of typical attacks

- Injecting malicious query (1) – Tautology



name:

pass:



```
$query = "SELECT * FROM users WHERE name=' X' OR 1=1; -- ' AND pass=' 1q2w3e4r '";  
$result = mysqli_query($conn, $query);
```



Table: users

id	name	pass	age
1	Alice	1q2w3e4r	23
2	Bob	pa\$\$w0rd	27

SQLi – Anatomy of typical attacks

- Injecting malicious query (1) – Tautology



name:

pass:

Condition `name='X' OR 1=1` always evaluates to `true`

Whatever that follow `--` are treated as `comments`



```
$query = 'SELECT * FROM users WHERE name=' X' OR 1=1; -- ' AND pass=' 1q2w3e4r '";  
$result = mysql_query($conn, $query);
```



Table: users

id	name	pass	age
1	Alice	1q2w3e4r	23
2	Bob	pa\$\$w0rd	27

Executed query: `SELECT * FROM users;`
→ All records in the table are returned!

SQLi – Anatomy of typical attacks

- Injecting malicious query (2) – Piggybacked queries



name:

pass:



```
$query = "SELECT * FROM users WHERE name='X'; DROP TABLE users; -- ' AND pass='...'";  
$result = mysqli_query($conn, $query);
```



Table: users

id	name	pass	age
1	Alice	1q2w3e4r	23
2	Bob	pa\$\$w0rd	27

SQLi – Anatomy of typical attacks

- Injecting malicious query (2) – Piggybacked queries



name:

pass:

First query

Second query

Comments



```
$query = "SELECT * FROM users WHERE name=' X' ; DROP TABLE users; -- ' AND pass=' ... '";  
$result = mysql_query($conn, $query);
```



Table: users

id	name	pass	age
1	Alice	1q2w3e4r	23
2	Bob	pa\$\$w0rd	27

Table is dropped!

SQLi – Anatomy of typical attacks

- Injecting malicious query (3) – Inference



name:

pass:



```
$query = "SELECT * FROM users WHERE name='X'; ORDER BY 9; -- ' AND pass='...'";  
$result = mysqli_query($conn, $query);
```



Table: users			
id	name	pass	age
1	Alice	1q2w3e4r	23
2	Bob	pa\$\$w0rd	27

MySQL Error: Unknown column 9 in order clause

SQLi – Anatomy of typical attacks

- Injecting malicious query (3) – Inference



name:

pass:



```
$query = "SELECT * FROM users WHERE name='X'; ORDER BY 5; -- ' AND pass='...'";  
$result = mysqli_query($conn, $query);
```



Table: users

id	name	pass	age
1	Alice	1q2w3e4r	23
2	Bob	pa\$\$w0rd	27

MySQL Error: Unknown column 5 in order clause

SQLi – Anatomy of typical attacks

- Injecting malicious query (3) – Inference



name:

pass:



```
$query = "SELECT * FROM users WHERE name='X'; ORDER BY 4; -- ' AND pass='...'";  
$result = mysqli_query($conn, $query);
```



Table: users

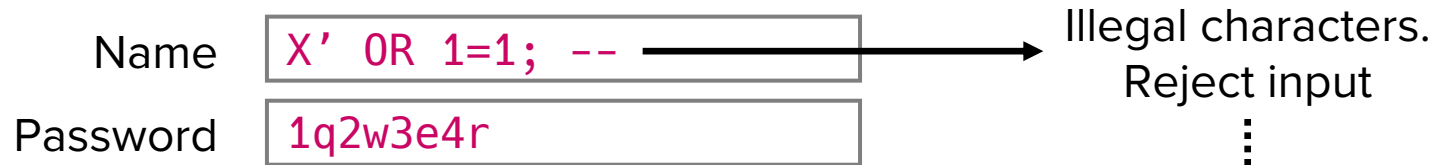
id	name	pass	age
1	Alice	1q2w3e4r	23
2	Bob	pa\$\$w0rd	27

We can infer that the table has four columns

No MySQL error (returns empty rows)

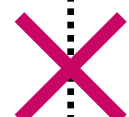
SQLi countermeasures

- Defense #1: Input sanitization
 - Option 1: **Disallow** special characters (e.g., `-=' ;`)



Not a user-friendly solution!

- What if legitimate name or pass include special chars?

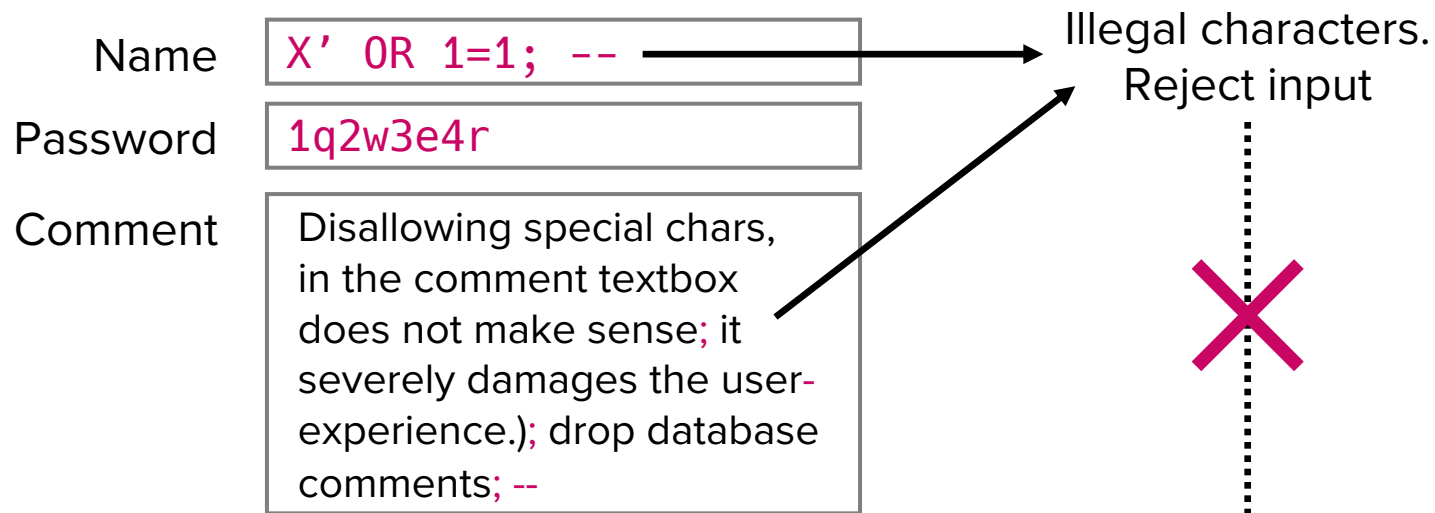


```
$query = "SELECT * FROM users WHERE name='$_GET["name"]' AND pass='$_GET["pass"]'";
```

SQLi countermeasures

- Defense #1: Input sanitization

- Option 1: **Disallow** special characters (e.g., `-=' ;`)



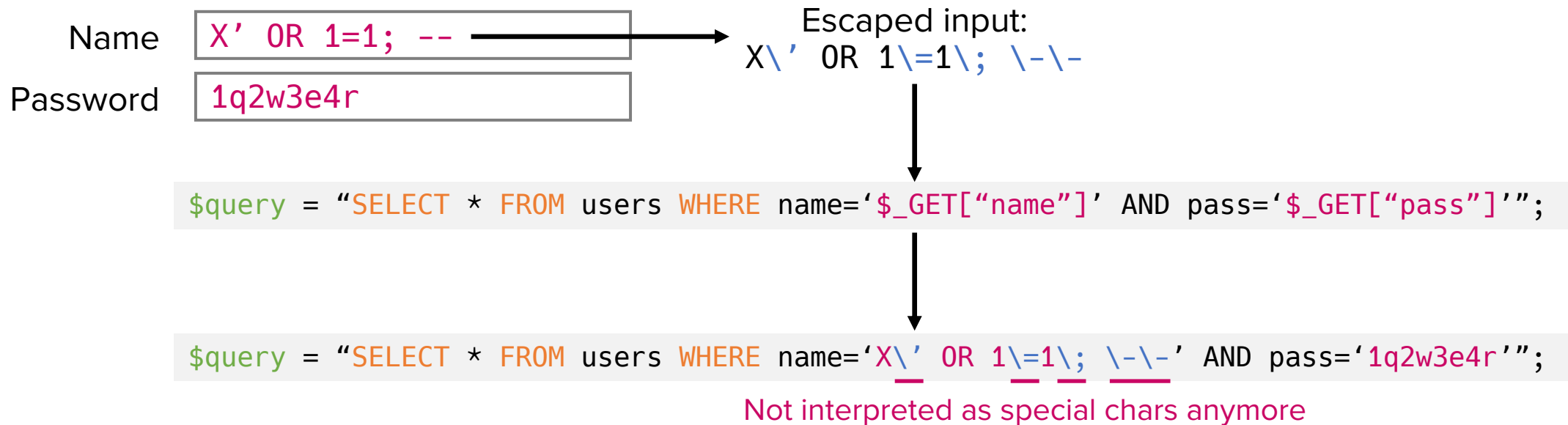
Not a user-friendly solution!

- What if legitimate name or pass include special chars?
- How to handle textboxes for long plaintext inputs that would normally contain special chars?

```
$query = "SELECT * FROM users WHERE name='$_GET["name"]' AND pass='$_GET["pass"]'";
```

SQLi countermeasures

- Defense #1: Input sanitization
 - Option 2: **Escape** special characters
 - Character escaping: Prepending a backslash (\) to a character to treat it as data, not special control character



SQLi countermeasures

- Defense #1: Input sanitization
 - Option 2: **Escape** special characters
 - Note: Never try to implement an escaper yourself!
 - Challenging to cover all edge cases
 - Prone to mistakes
 - e.g., `string.replace("'", "\'")` fails if `string = "'\''\''\''"`
 - Utilize existing functions
 - `mysqli_real_escape_string` of PHP

SQLi countermeasures

- Defense #2: Prepared statements
 - Represent dynamic data with question marks (?)
 - Database can always distinguish code and data

```
$query = "SELECT * FROM users WHERE name='$_GET["name"]' AND pass='$_GET["pass"]'";
```

 Vulnerable!

```
$query = $db->prepare("SELECT * FROM users WHERE name=? AND pass=?");  
$query->bind_param("ss", $_GET["name"], $_GET["pass"]);
```

 Safer!

DB encodes the parameterized data → no risk of misinterpreting data as code

Tell DB what type of data to expect (s: string, i: integer, ...)

SQL Injection IRL

SQLi in real life?



High-impact SQLi vulnerabilities

- Some tools or templates are reused by many people
 - Example: WordPress for website building
 - Vulnerabilities included in these heavily-reused tools often **propagate** to a number of websites

WordPress

- Simplified website building procedure
 - Select a website template
 - Customize components (e.g., title, font, menu items, ...)
 - Add plugins for additional features
 - User authentication
 - Credit card checkout
 - PayPal payments
 - reCAPTCHA
 - Private messaging
 - ...

Issues in WordPress

- WP Private Messages plugin

```
// wpu_private_messages.php
function wpu_read() {
    global $wpdb, $current_user, $wpulang;
    $id = $_GET["id"];
    $r = $_GET["r"];
    // ...
    $pm = $wpdb->get_row(
        "SELECT * FROM $wpdb->prefix".private_messages." WHERE id = $id", ARRAY_A
    );
    // ...
}
```

Looks familiar? :)

Issues in WordPress

- CVE-2024-2879 in LayerSlider plugin
 - Creates smooth slide effects for popups
 - More than a million active installations

```
function ls_get_popup_markup() {  
    $id = is_numeric( $_GET['id'] ) ? (int) $_GET['id'] : $_GET['id'];  
    $popup = LS_Sliders::find( $id ); // to identify popup object  
    // ...  
}
```



```
public static function find($args = array()) {  
    if (is_numeric($args) && intval($args) == $args) {  
        return self::_getById( (int) $args );  
    } else {  
        /* next page! */  
    }  
}
```

Issues in WordPress

- CVE-2024-2879 in LayerSlider plugin
 - Creates smooth slide effects for popups

```
// continued from the previous page
$where = "WHERE {$args['where']} "; // Looks familiar? :)
$sliders = $wpdb->get_results("
    SELECT SQL_CALC_FOUND_ROWS {$args['columns']}
    FROM $table
    $where
    ORDER BY `{$args['orderby']}` `{$args['order']}`, name ASC
    LIMIT {$args['limit']}", ARRAY_A);
// ...
```

Example attack payload:

```
id[where]=(SELECT * FROM (SELECT SLEEP(3600)) haha)
```

Inference Attack

Background: Fine-grained access control

- DB servers implement record-granularity access control
 - Authenticated users can access specific rows with access rights
 - Example: User can only retrieve his/her own exam score

```
def get_score(user):  
    q = "select score from students;"  
    results = execute(q)  
    for record in results:  
        if record.sid == user.sid:  
            return record
```

sid	name	year	score
201805	Alice	senior	77
201733	Bob	junior	96
200216	Claire	freshman	45
200218	Dave	sophomore	85
201909	Eve	junior	68

Background: SQL aggregate functions

- Aggregate functions in SQL calculates a single result from multiple values
 - Perform calculation directly on the raw database (fast!)
 - sum, avg, count, min, max

```
SELECT SUM(score) FROM students;
```

→ 371

```
SELECT COUNT(*) FROM students;
```

→ 5

sid	name	year	score
201805	Alice	senior	77
201733	Bob	junior	96
200216	Claire	freshman	45
200218	Dave	sophomore	85
201909	Eve	junior	68

DB inference attack

- Inferring sensitive data from non-sensitive data
 - Subtle vulnerability in statistical databases
 - Two types of inference attacks:
 - Statistical attacks
 - Functional dependency attacks

sid	name	year	score
201805	Alice	senior	77
201733	Bob	junior	96
200216	Claire	freshman	45
200218	Dave	sophomore	85
201909	Eve	junior	68

DB inference attack

- Inferring sensitive data from non-sensitive data
 - Type 1: Statistical attack use aggregate functions

```
SELECT AVG(score)
FROM students
WHERE year="junior"; → 82
```

“Junior students’ average exam score is 82.”

→ Non-sensitive data

Bob knows his score is 96.

Bob also knows the only other junior is Eve.

→ Bob can infer Eve’s score, which is sensitive data

sid	name	year	score
201805	Alice	senior	77
201733	Bob	junior	96
200216	Claire	freshman	45
200218	Dave	sophomore	85
201909	Eve	junior	68

DB inference attack

- Inferring sensitive data from non-sensitive data
 - Type 2: Functional dependency attack use multiple queries
 - FD: $A \rightarrow B$
 - Any two rows in a table that have the same value of A must have the same value of B
 - Example FD: Rank \rightarrow Salary
 - Non-sensitive: (name, rank), (rank, salary)
 - Sensitive: (name, salary)
 - Can be inferred from FD: Rank \rightarrow Salary

Inference countermeasures

- Query restriction: Disallow aggregate functions if the number of selected records are either too small or too large
 - For a table of size N , a query is permitted if the number of matching records for an aggregate function, i.e., C , satisfies $k \leq C \leq N - k$
 - k is a threshold set by system
 - Querying the **AVG** of one person is denied if threshold $k > 1$
 - Why do we need an upper bound?
 - Querying the **AVG** of $N - 1$ people reveals the value of the remaining person!

Compromise availability for confidentiality

Inference countermeasures

- **Perturbation: Provide approximate answers to queries**
 - Add noise to the statistics generated from the original data
 - e.g., AVG is 84.5 instead of 82 (correct average)
- **Result clustering: Provide a range rather than precise answers**
 - User only gets in the ballpark and cannot infer others' records
 - e.g., AVG is 70~90

Both methods compromise precision for confidentiality

Summary

- DB is a crucial backend for most web-based services
- SQL injection attacks happen while interfacing the front-end and the back-end
 - Never trust user inputs!
- Inference attack is inevitable
 - Mitigation is very difficult

Questions?