Lec 24: Database Security

CSED415: Computer Security

Spring 2025

Seulbae Kim

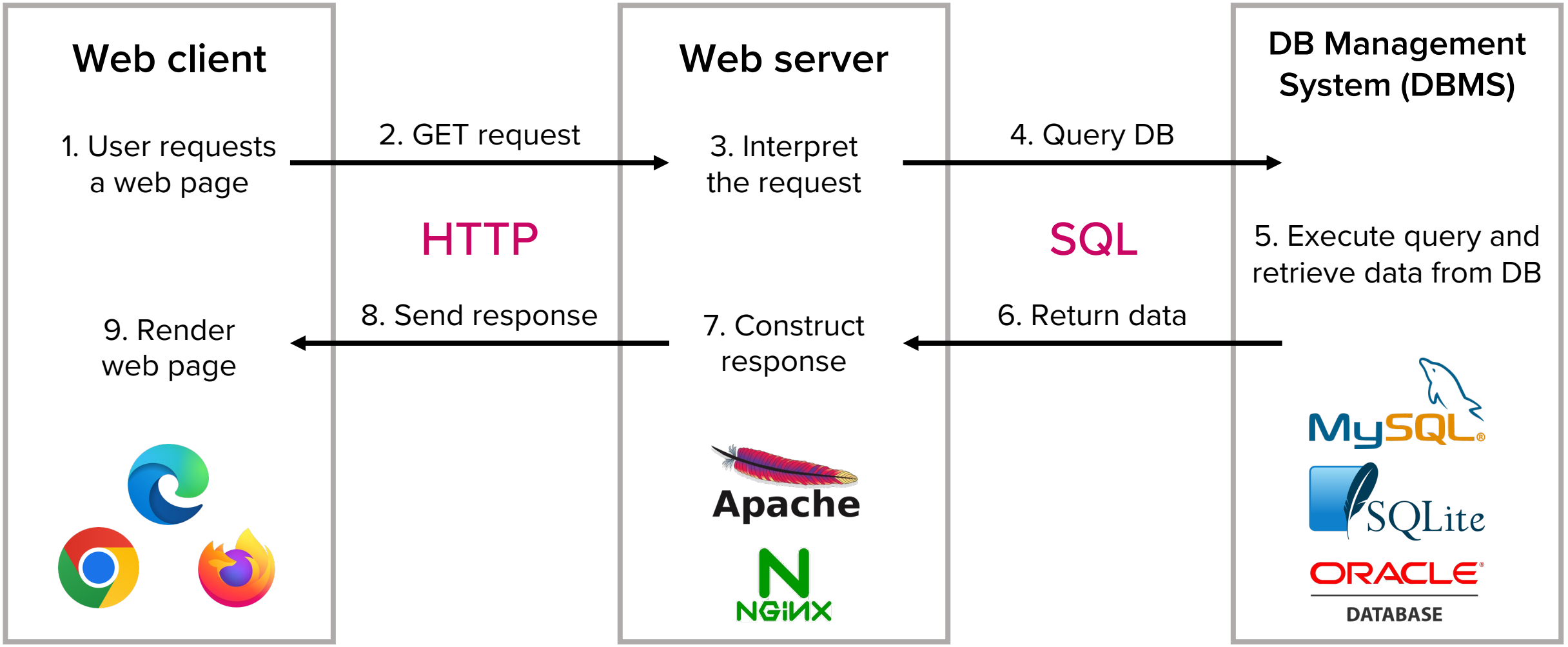
POSTECH
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

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

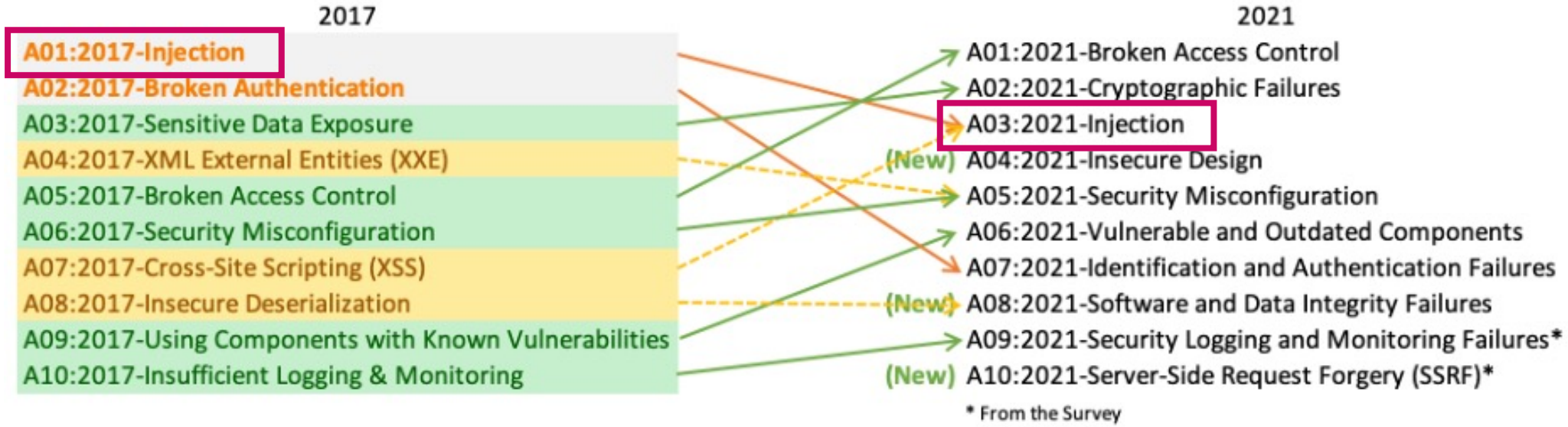
Most web applications rely on “database” systems

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

Attribute

Record

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

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 * FROM students;
```

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

Table: students

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

Table: students

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

Table: students

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

Table: students

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

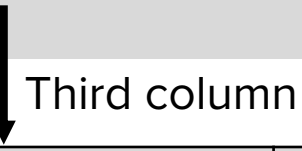
Table: students

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 3;
```



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

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]
  - 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);
```

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]
  - 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

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 – 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 to understand today's lecture

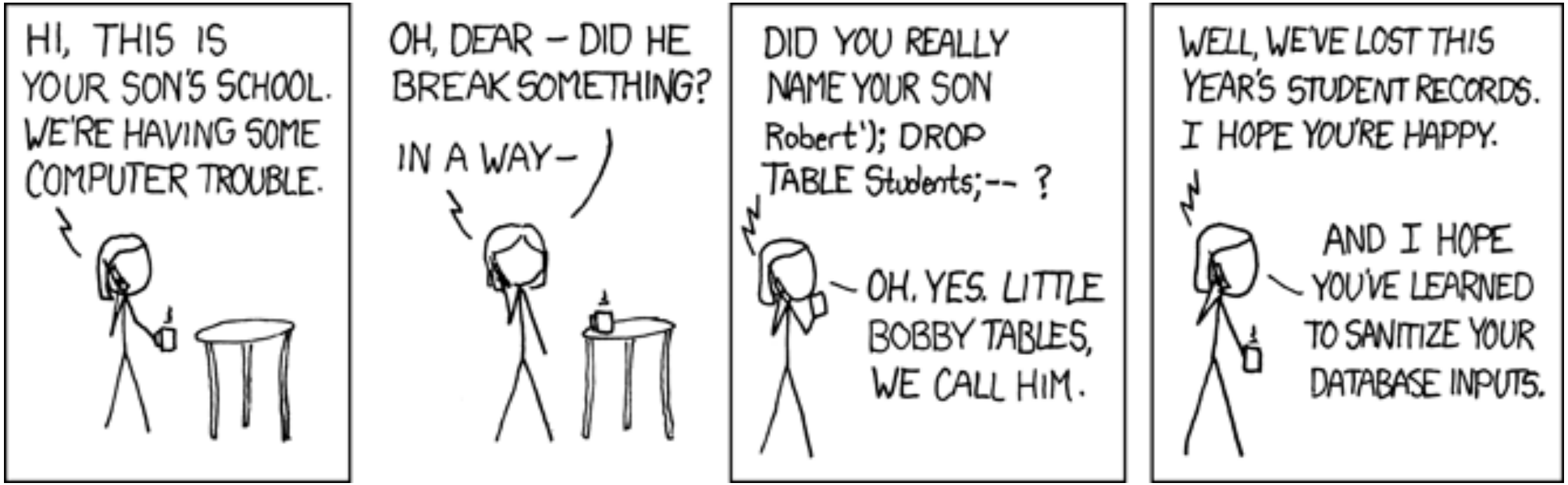
SQL Injection Attack

SQL injection (SQLi)

- SQLi: Injecting malicious SQL queries into server to cause unintended behavior of DBMS (DB management systems)
  - Typically caused by vulnerable string manipulation logic for constructing SQL queries
  - Allows attackers 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”

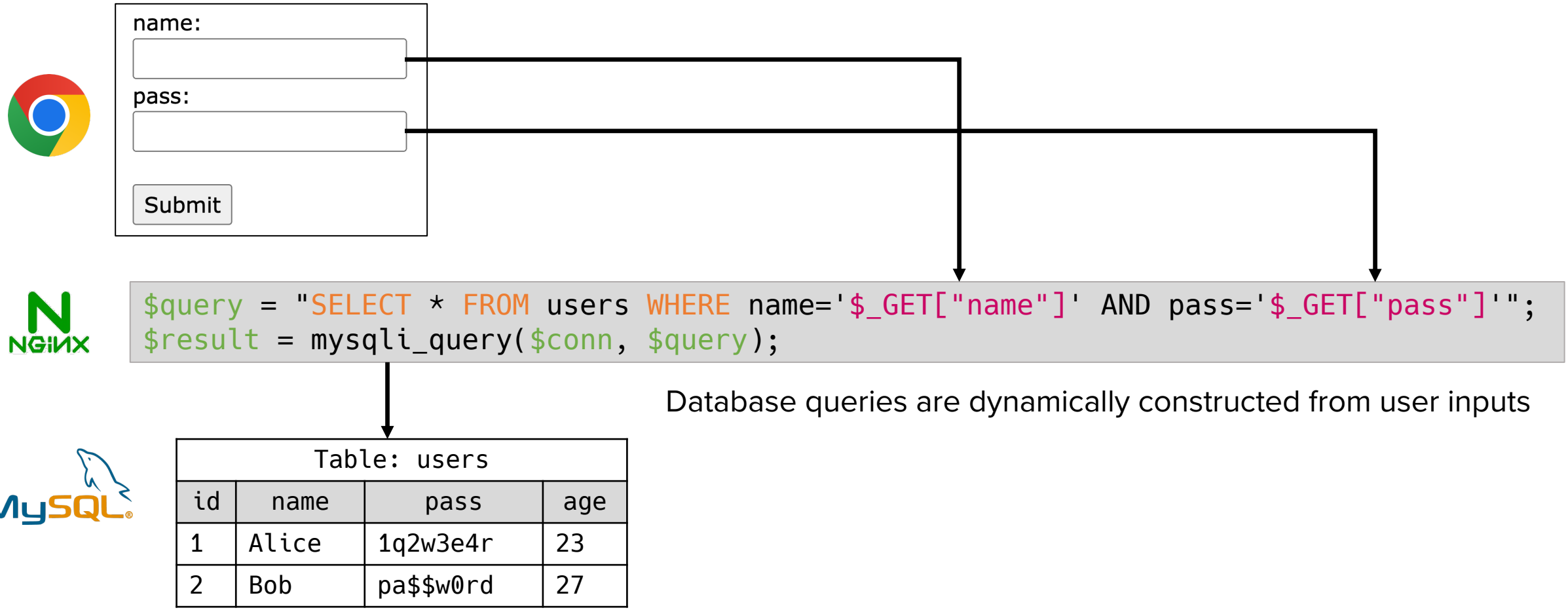


https://xkcd.com/327/

SQLi – Anatomy of typical attacks

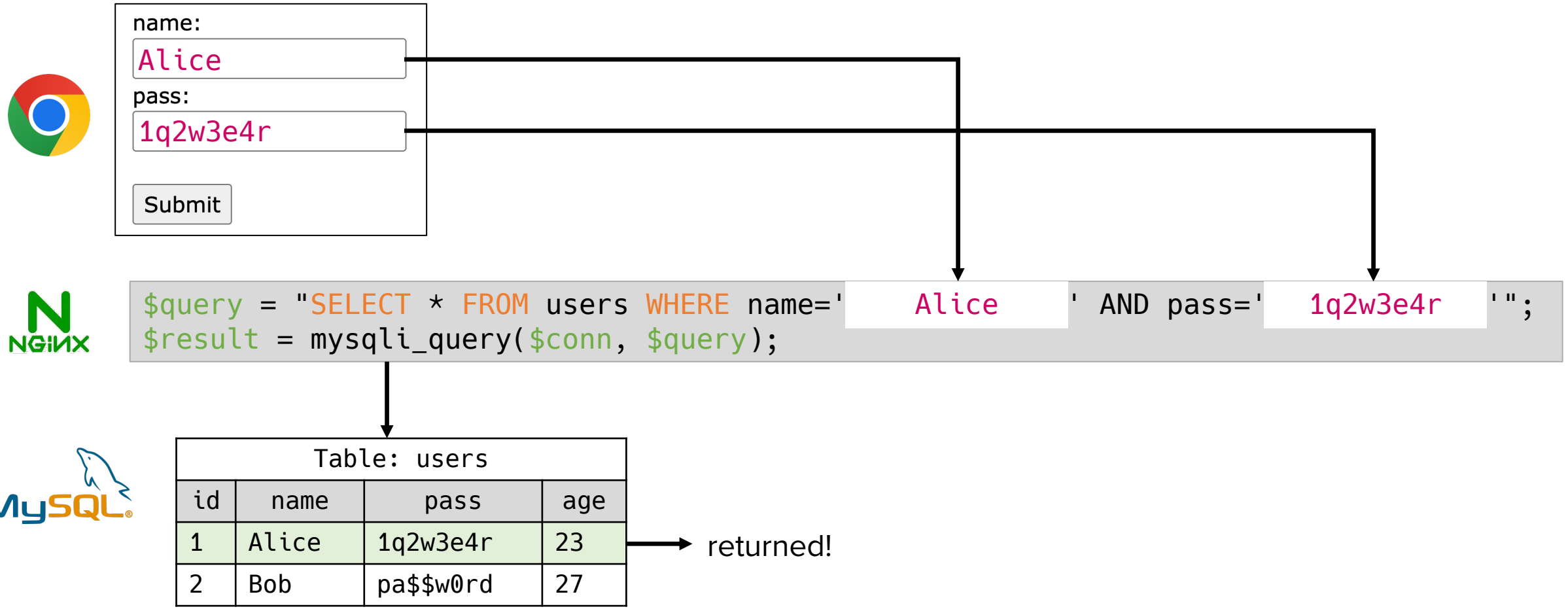
POSTECH

- A simple client-server interaction



SQLi – Anatomy of typical attacks

- A simple client-server interaction



SQLi – Anatomy of typical attacks

- Injecting malicious query (1) – Tautology

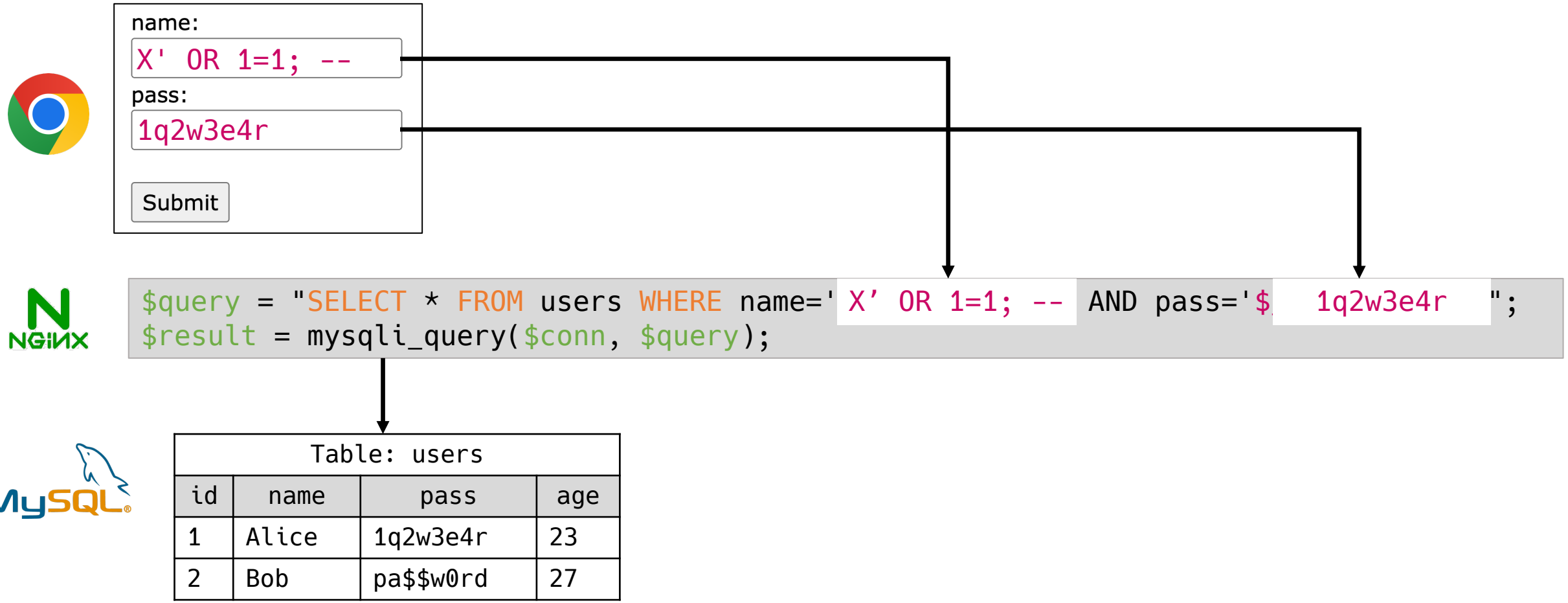


Table: users

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

SQLi – Anatomy of typical attacks

- Injecting malicious query (1) – Tautology

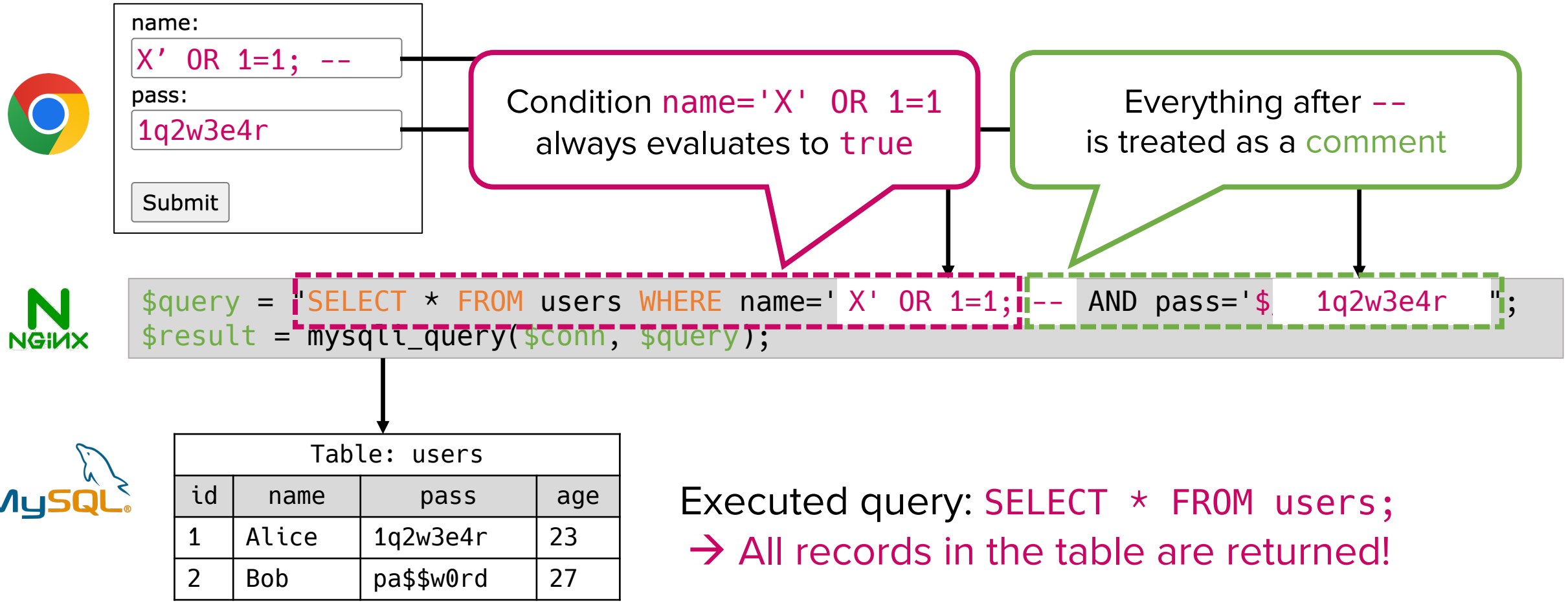


Table: users

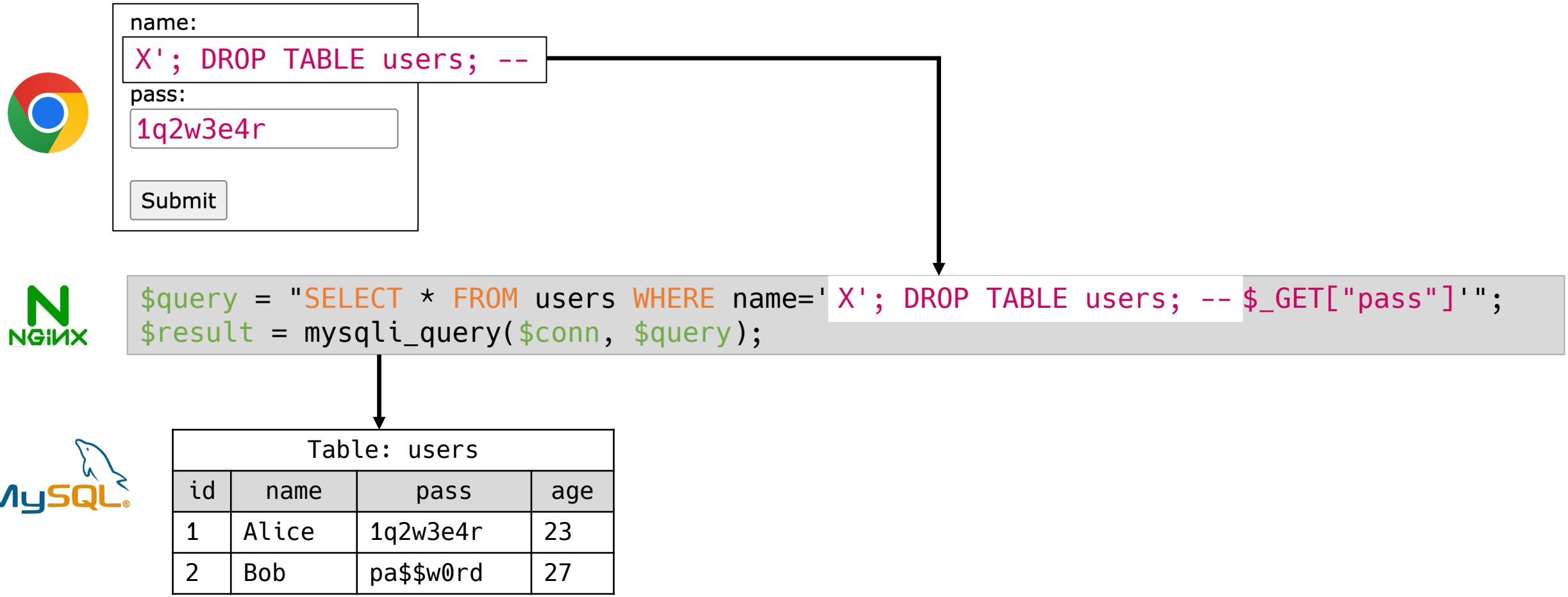
id	name	pass	age
1	Alice	1q2w3e4r	23
2	Bob	pa$$w0rd	27

Executed query: `SELECT * FROM users;`

→ All records in the table are returned!

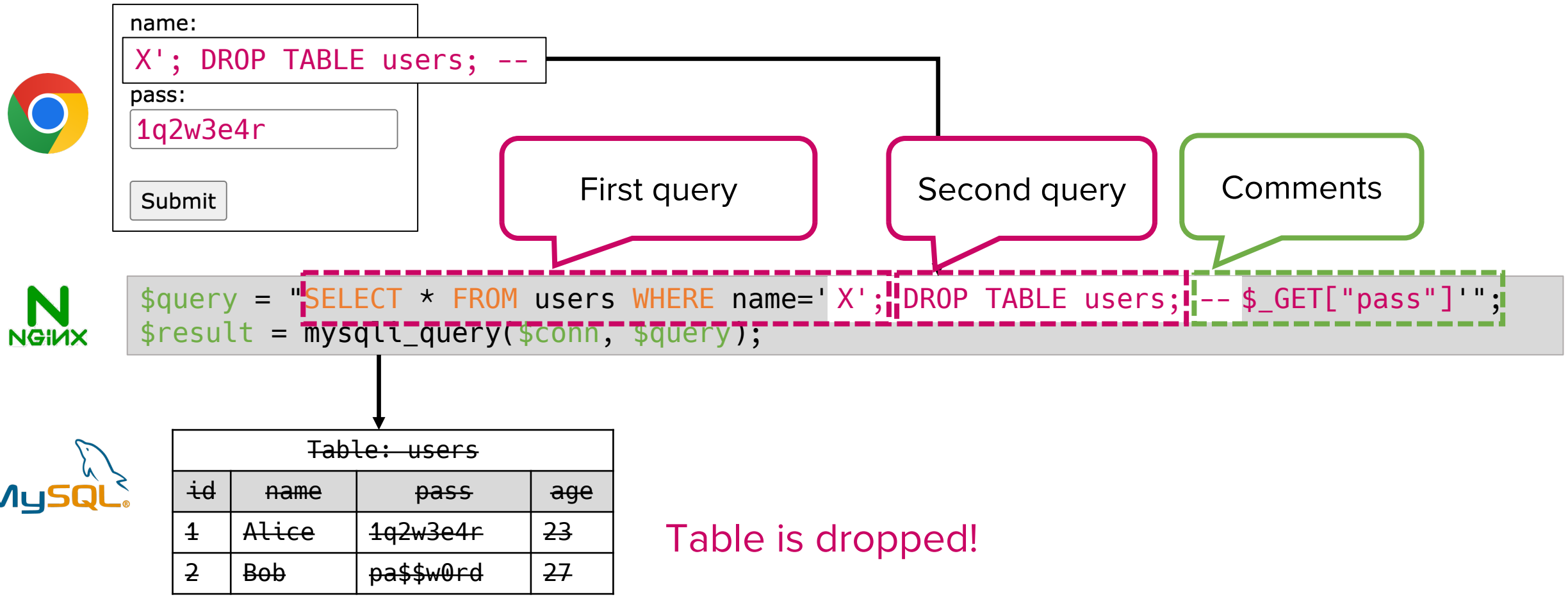
SQLi – Anatomy of typical attacks

- Injecting malicious query (2) – Piggybacked queries



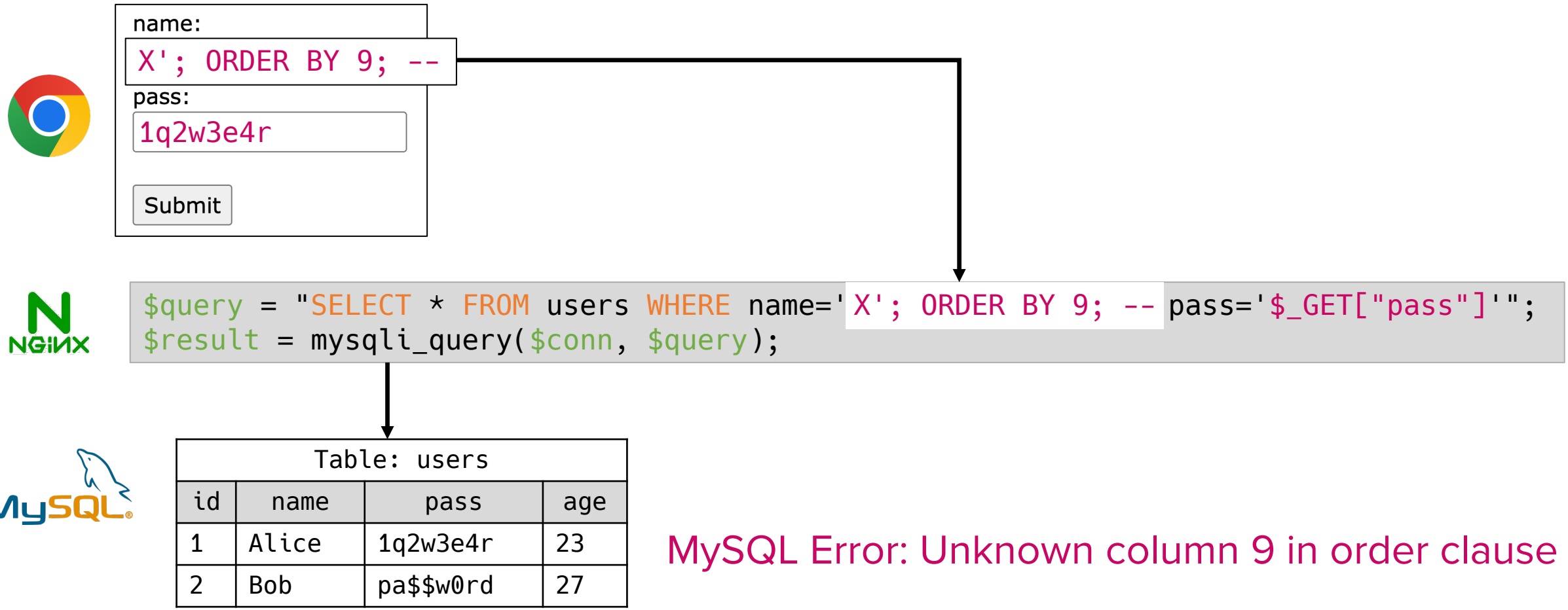
SQLi – Anatomy of typical attacks

- Injecting malicious query (2) – Piggybacked queries



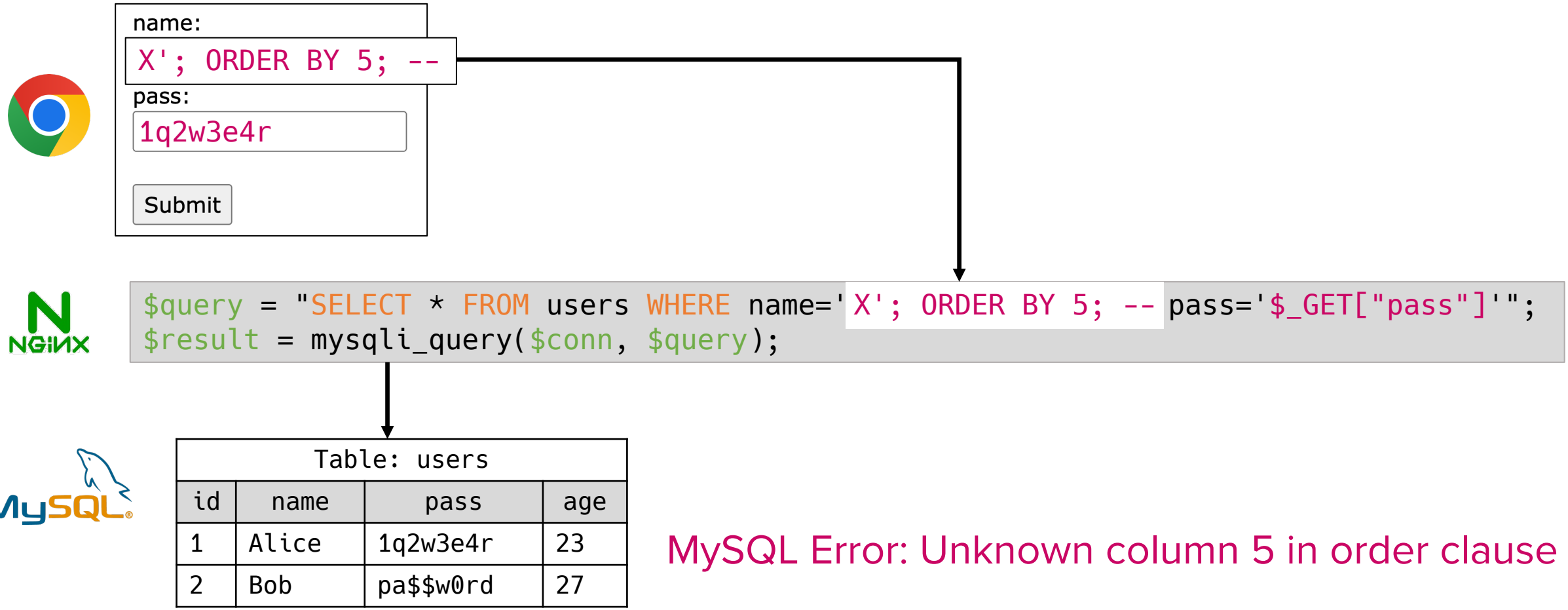
SQLi – Anatomy of typical attacks

- Injecting malicious query (3) – Inference



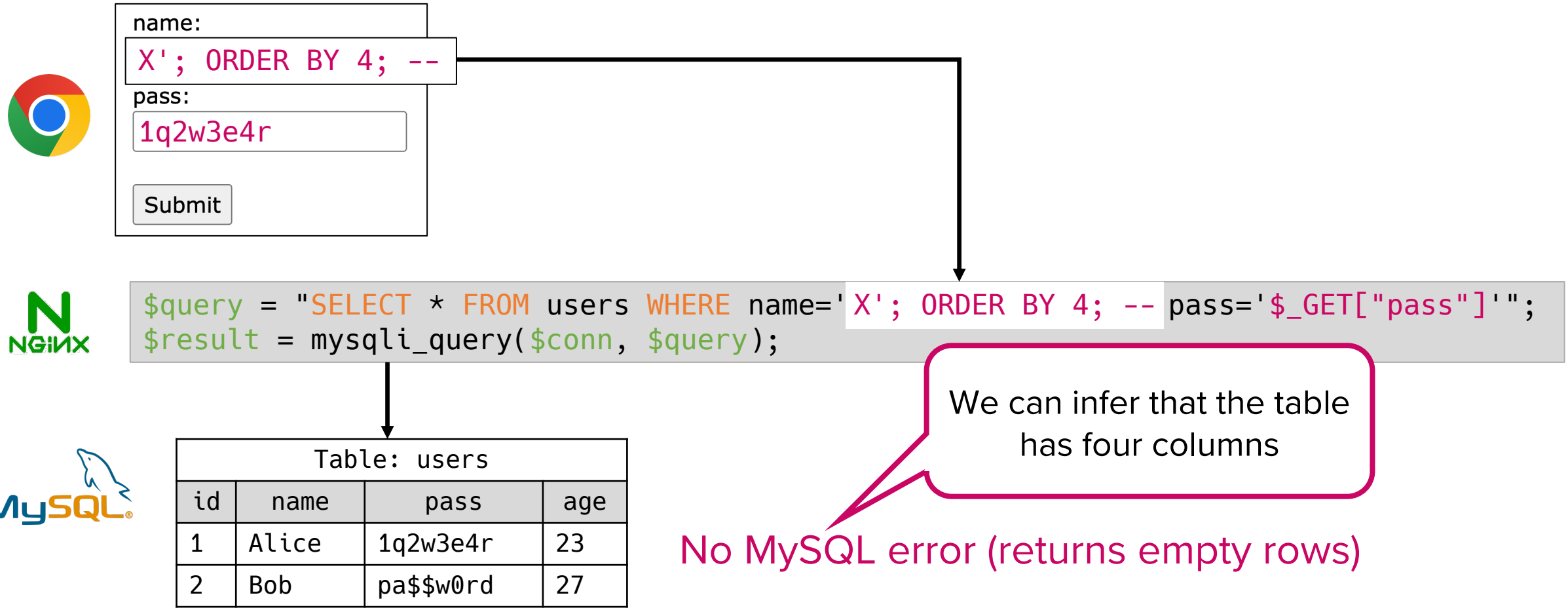
SQLi – Anatomy of typical attacks

- Injecting malicious query (3) – Inference



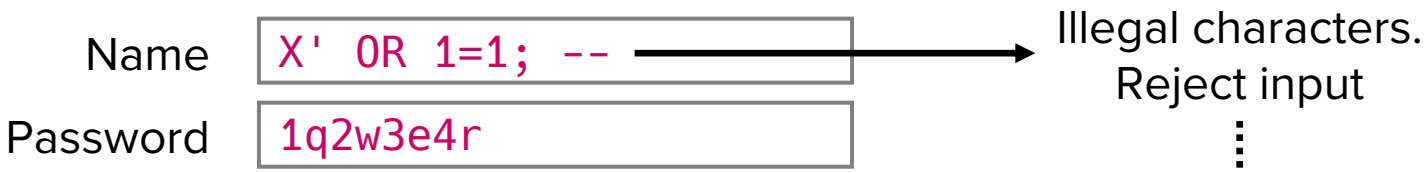
SQLi – Anatomy of typical attacks

- Injecting malicious query (3) – Inference



SQLi countermeasures

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



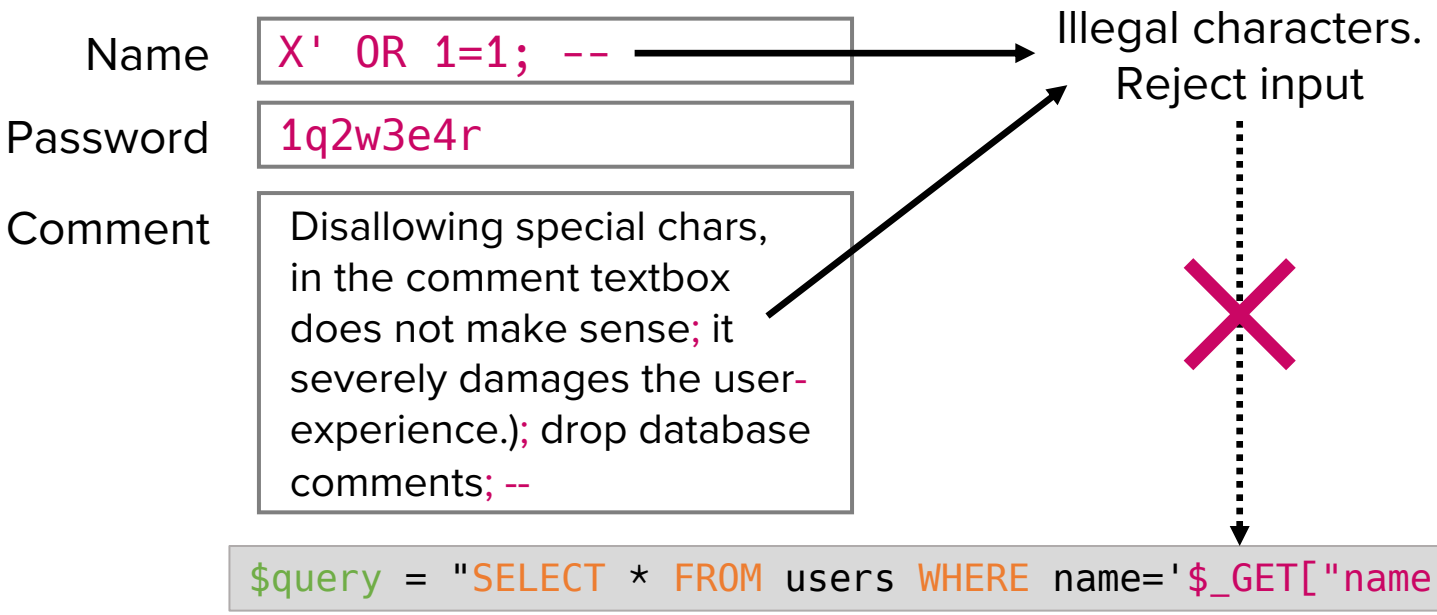
Illegal characters.
Reject input

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., `-='";`)



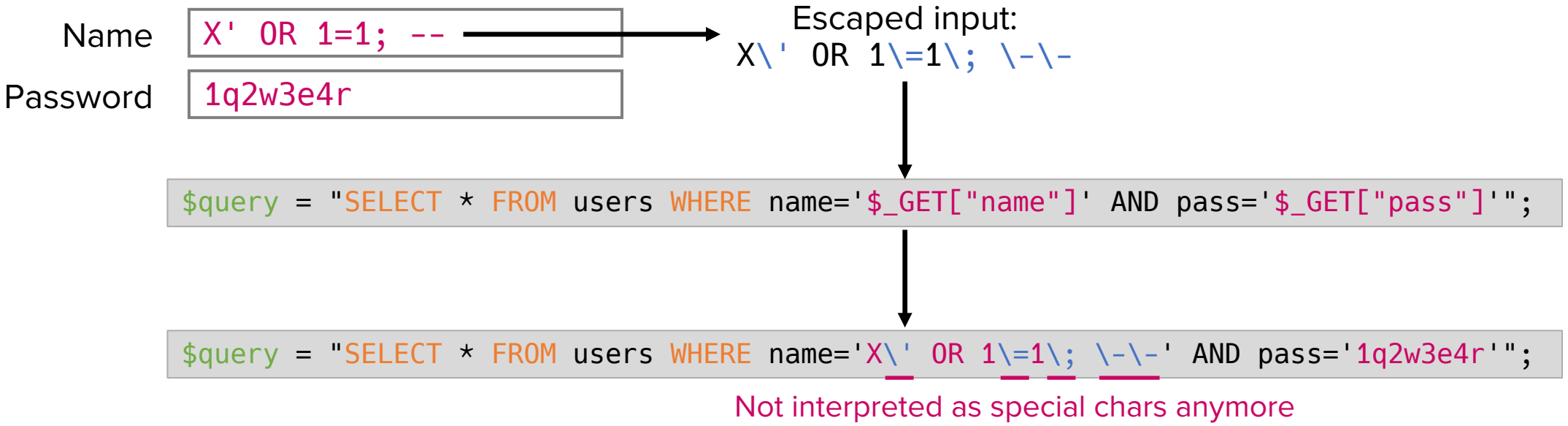
Not a user-friendly solution!

- 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 = "'\'\\'\'"`
    - We can utilize existing, heavily-tested functions
      - e.g., `mysqli_real_escape_string` of PHP

SQLi countermeasures

POSTECH

- 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!

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

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

Real-World SQLi Vulnerabilities

Real-world SQLi?

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

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

Security issues found in WordPress

- WP Private Messages plugin
  - Allow site users to send private messages to each other
  - Vulnerability in message read function:

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

Looks familiar? :)

Security issues found in WordPress

- CVE-2022-0651 in WP Statistics
  - Visitor analytics plugin (like Google Analytics)
  - More than 600,000 active installations

```
$current_page = array(
  'type' => $this->rest_hits->current_page_type,
  'id' => $this->rest_hits->current_page_id,
);
// ...
$exist = $wpdb->get_row(
  "SELECT `page_id` FROM `" . DB::table('pages') .
  "` WHERE `date` = '" . TimeZone::getCurrentDate('Y-m-d') . "' " .
  // ...
  "AND `type`='{$current_page['type']}' AND `id`={$current_page['id']}", ARRAY_A
);
```

Looks familiar? :)

Security issues found in WordPress

- CVE-2022-0651 in WP Statistics

```
Attack payload: HTTP request to URL/wp-json/wp-
statistics/v2/hit?_=11&_wpnonce=935551c012&wp_statistics_hit_rest=&browser=&platform=&version=&referred=&
ip=11.11.11.11&exclusion_match=no&exclusion_reason&ua=Something&track_all=1&timestamp=11&current_page_typ
e=home&current_page_id=sleep(30)&search_query&page_uri=/&user_id=0
```

```php
$current_page = array(
  'type' => $this->rest_hits->current_page_type,
  'id' => $this->rest_hits->current_page_id,
);
// ...
$exist = $wpdb->get_row(
  "SELECT `page_id` FROM `" . DB::table('pages') .
  "` WHERE `date` = '" . TimeZone::getCurrentDate('Y-m-d') . "' " .
  // ...
  "AND `type`='{$current_page['type']}' AND `id`={$current_page['id']}", ARRAY_A
);
```

sleep(30)

* In PHP, {sleep(30)} executes sleep(30);

Security issues found in WordPress

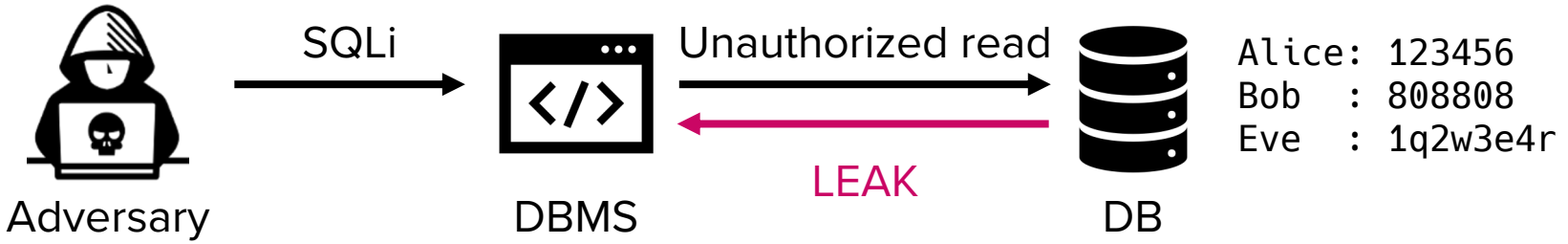
POSTECH

- CVE-2022-0651 in WP Statistics
  - Patched by sanitizing and forcing type
  - Patch:

```
$current_page = array(
  'type' => esc_sql($this->rest_hits->current_page_type),
  'id' => esc_sql($this->rest_hits->current_page_id),
);
// ...
$exist = $wpdb->get_row(
  "SELECT `page_id` FROM `" . DB::table('pages') .
  "` WHERE `date` = '" . TimeZone::getCurrentDate('Y-m-d') . "' " .
  // ...
  "AND `type`='{$current_page['type']}' AND `id`='{$current_page['id']}'", ARRAY_A
);                                                    '{sleep(30)}' is a string
```

Other incidents

- Yahoo voices (2012)
  - SQLi vulnerability → DB leaked → Passwords were stored in plaintext
  - 450,000 accounts affected
- Zynga mobile game server (2019)
  - SQLi → DB leaked → Passwords were stored in unsalted MD5
  - 170 million accounts affected



Inference Attack

Background: Fine-grained access control

POSTECH

- 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
```

Table: students			
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

Table: students

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

Table: students			
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

Table: students

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 → B
    - Any two rows in a table that have the same value of A must have the same value of B
    - Example FD: Rank → Salary
  - Non-sensitive: (name, rank), (rank, salary)
  - Sensitive: (name, salary)
    - Can be inferred from FD: Rank → 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 [80:90]

Both countermeasures inevitably compromise precision for confidentiality

Summary

- Databases are essential backends for most web-based services
- SQL injection attacks exploit the interface between the front end and the database
  - Defense: Never trust user input. Always validate and sanitize
- Inference attacks can be difficult to detect and often unavoidable

Questions?