
WEEK 2 – DAY 8

INJECTION ATTACKS (SQL & NoSQL)

1. WHAT IS AN INJECTION ATTACK? (CORE IDEA)

Simple Definition

Injection happens when **untrusted user input** is treated as **executable code** instead of data.

The application:

1. Accepts user input
2. Builds a query or command using that input
3. Executes it **without proper validation or separation**

An attacker **injects malicious logic** into that input.

2. WHY INJECTION IS DANGEROUS

Injection can lead to:

- Authentication bypass
- Data theft (users, passwords, credit cards)
- Data modification or deletion
- Complete system compromise

Historical context:

- Injection has been **OWASP Top 10 #1 multiple times**
 - Many real-world breaches started with a single injectable parameter
-

3. SQL INJECTION (SQLi)

What Is SQL Injection?

SQL Injection occurs when **SQL queries are constructed by concatenating strings with user input**.

Vulnerable Example (Login)

```
query = "SELECT * FROM users WHERE username = '" + username + "' AND password = '" + password  
+ "'"
```

If user enters:

username: admin

password: ' OR '1'='1

Final query becomes:

```
SELECT * FROM users WHERE username = 'admin' AND password = " OR '1'='1'
```

This condition is **always true** → login bypass.

4. TYPES OF SQL INJECTION

1. Classic SQL Injection

- Injects SQL logic directly
 - Example: ' OR 1=1 --
-

2. Union-Based SQL Injection

Attacker extracts data using UNION.

```
' UNION SELECT username, password FROM users --
```

3. Blind SQL Injection

- No visible error or output
- Attacker infers data via:
 - True/false responses
 - Time delays (SLEEP())

Example:

```
' AND IF(1=1, SLEEP(5), 0) --
```

5. NoSQL INJECTION (MongoDB Example)

Why NoSQL Is Also Vulnerable

Developers wrongly assume:

“No SQL means no injection”

This is **false**.

Vulnerable MongoDB Query (Python)

```
db.users.find_one({
```

```
"username": username,  
"password": password  
})
```

If attacker sends:

```
{  
  "username": {"$ne": null},  
  "password": {"$ne": null}  
}
```

Query becomes:

Find any user where username is not null AND password is not null

→ Authentication bypass.

6. COMMON NoSQL INJECTION TECHNIQUES

- \$ne (not equal)
- \$gt, \$lt
- \$or
- Regex abuse

Example:

```
{"username": {"$regex": ".*"}}
```

7. HANDS-ON: EXPLOITING SQL INJECTION

Step 1: Identify Input

- Login forms
 - Search fields
 - URL parameters
-

Step 2: Test for Injection

Try:

```
' OR 1=1 --
```

If login succeeds → vulnerable.

Step 3: Observe Errors

SQL error messages:

- MySQL
- PostgreSQL
- SQLite

Errors reveal:

- Table names
 - Column names
 - DB type
-

8. PREVENTION STRATEGIES (MOST IMPORTANT)

1. Parameterized Queries (BEST DEFENSE)

Secure SQL Example (Python)

```
cursor.execute(  
    "SELECT * FROM users WHERE username = ? AND password = ?",
    (username, hashed_password)  
)
```

Why this works:

- Query structure is fixed
 - Input is treated as data, not code
-

2. Prepared Statements

- SQL is precompiled
 - Parameters are bound later
-

3. Input Validation (SECONDARY)

- Type checks
- Length limits
- Whitelisting

 Never rely on validation alone.

4. ORM Usage

- SQLAlchemy
- Django ORM

ORMs automatically parameterize queries **if used correctly.**

5. Least Privilege (Defense in Depth)

- App user should not be DB admin
 - Prevents damage even if injection exists
-

9. FIXING NoSQL INJECTION

BAD

```
db.users.find_one(request.json)
```

GOOD

```
db.users.find_one({  
    "username": str(username),  
    "password": str(password)  
})
```

Also:

- Disable operator injection
 - Validate input schema
 - Use strict typing
-

10. INTERVIEW QUESTIONS & MODEL ANSWERS

Q1: What is SQL Injection?

“SQL Injection is a vulnerability where user input is improperly embedded into SQL queries, allowing attackers to manipulate query logic and execute unauthorized operations.”

Q2: Why parameterized queries prevent SQL Injection?

“Because they separate query structure from user input, ensuring input is treated strictly as data.”

Q3: Is NoSQL immune to injection?

"No. NoSQL databases are vulnerable to injection through query operators like \$ne and \$or if input is not validated."

Q4: ORM vs Raw SQL — which is safer?

"ORMs are safer by default, but misuse can still lead to injection."

11. SECURITY MINDSET (ATTACKER VS DEFENDER)

Attacker Thinks:

- Where does my input reach the database?
- Can I break query structure?

Defender Thinks:

- Is input ever treated as code?
- Can user input alter query logic?