# LESSON 32
# Security Basics (XSS, SQLi)

**WEEK 07**

# SQL Injection

# Introduction to SQL Injection

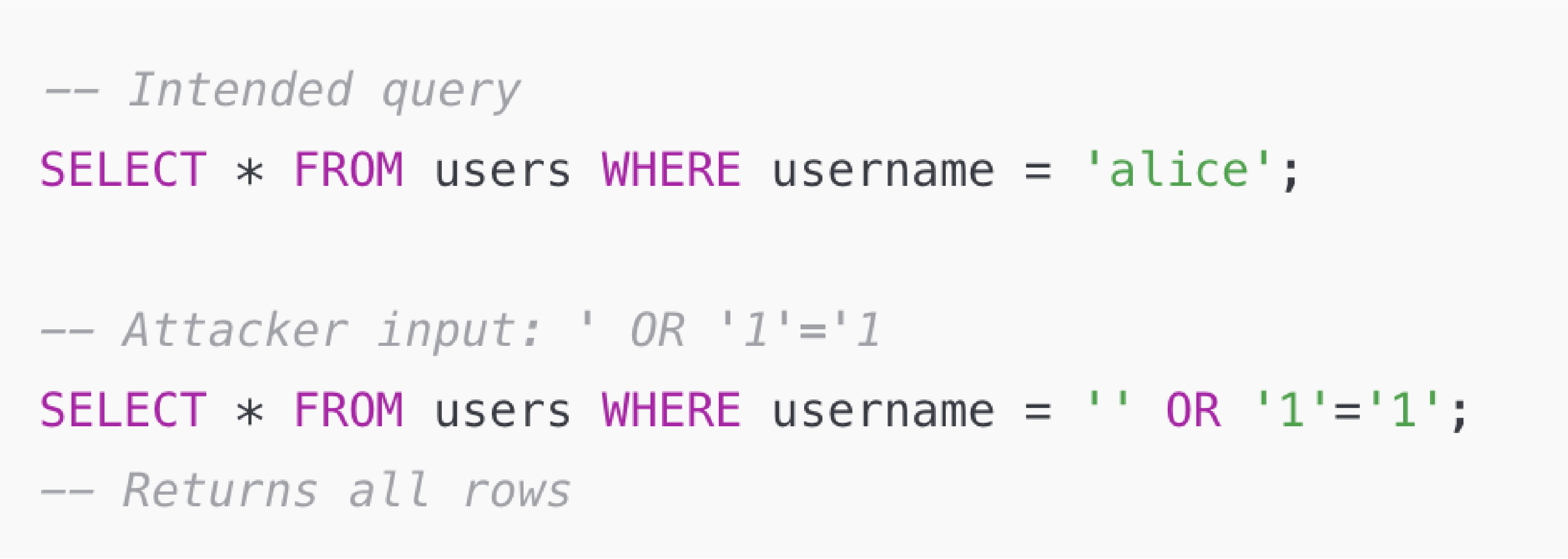
- ❖ Understand SQL Injection basics and how it impacts applications
- ❖ Learn how SQL Injection happens in Java Spring Boot
- ❖ Explore prevention strategies and best practices
- ❖ Implement secure database access patterns

# What is SQL Injection?

- A code injection technique targeting SQL queries
  - Occurs when untrusted input is used to dynamically build SQL statements
  - Allows attackers to manipulate queries, retrieve/modify data
  - One of the most common OWASP Top 10 vulnerabilities

# What is SQL Injection?

- ❖ A technique where an attacker manipulates SQL queries by injecting malicious code into input fields



```sql
-- Intended query
SELECT * FROM users WHERE username = 'alice';

-- Attacker input: ' OR '1'='1
SELECT * FROM users WHERE username = '' OR '1'='1';
-- Returns all rows
```

# Impact of SQL Injection

- Famous breaches:
  - Sony Pictures (2011),
  - Heartland Payment Systems (2008)
- Data theft, authentication bypass, financial loss
- Compliance violations (GDPR, PCI-DSS)

# How SQL Injection Works

## ❖ Attack flow

- ➢ Attacker injects malicious SQL via form fields, query parameters, headers
- ➢ Application concatenates input into SQL
- ➢ Database executes unintended commands
- ➢ Example: "SELECT * FROM users WHERE username = '" + userInput + "';"

# How SQL Injection Works

## ❖ Exploitation flow

- ➢ User input:

```
admin' --
```

- ➢ Server builds query:

```sql
SELECT * FROM users WHERE username = 'admin' -- ' AND password = 'pass';
```

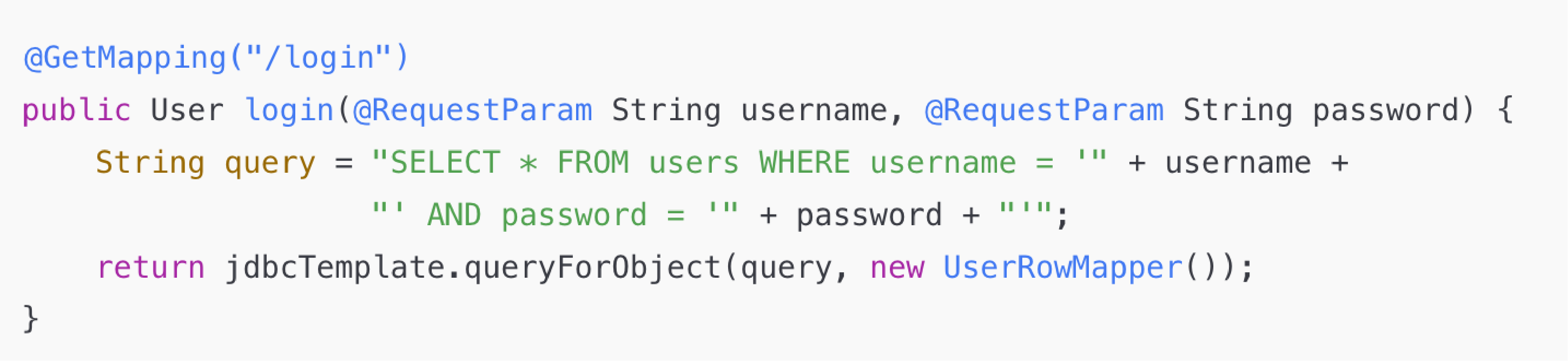
- ➢ Database executes without password verification

# SQL Injection in Java Spring Boot

- **Why Spring Boot apps are still vulnerable**
  - Spring Data JPA reduces risk but does not eliminate it
  - Native queries and dynamic JPQL can still be exploited
  - Legacy JDBC code often uses string concatenation

# Example Vulnerable Endpoint

- **Simple login form vulnerability**
  - Attacker can inject "' OR '1'='1" to bypass login



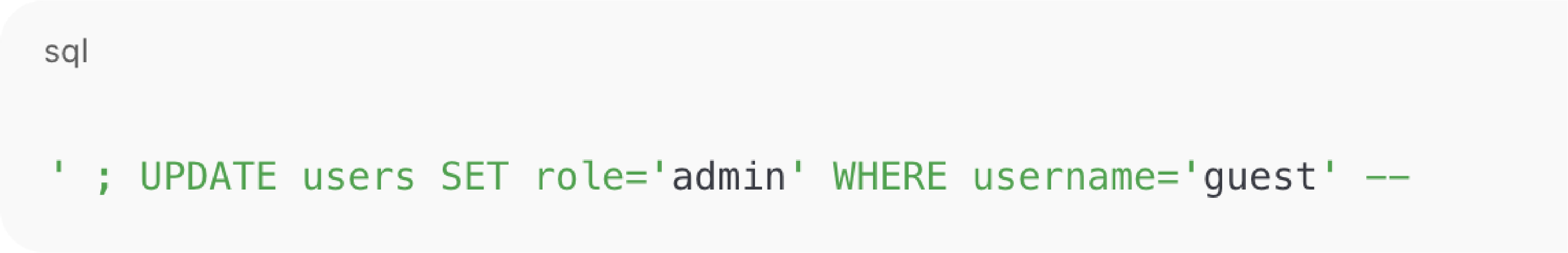
```java
@GetMapping("/login")
public User login(@RequestParam String username, @RequestParam String password) {
    String query = "SELECT * FROM users WHERE username = '" + username +
                   "' AND password = '" + password + "'";
    return jdbcTemplate.queryForObject(query, new UserRowMapper());
}
```

# Impact Assessment

- **What could happen if exploited**
  - Unauthorized access
  - Data modification/deletion
  - Privilege escalation

# Impact Assessment

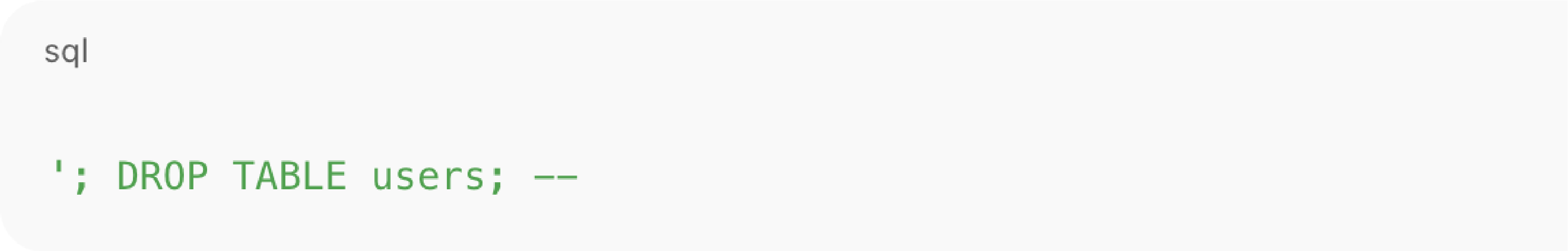
- Modify data:



sql

```sql
' ; UPDATE users SET role='admin' WHERE username='guest' --
```

- Drop tables:



sql

```sql
'; DROP TABLE users; --
```

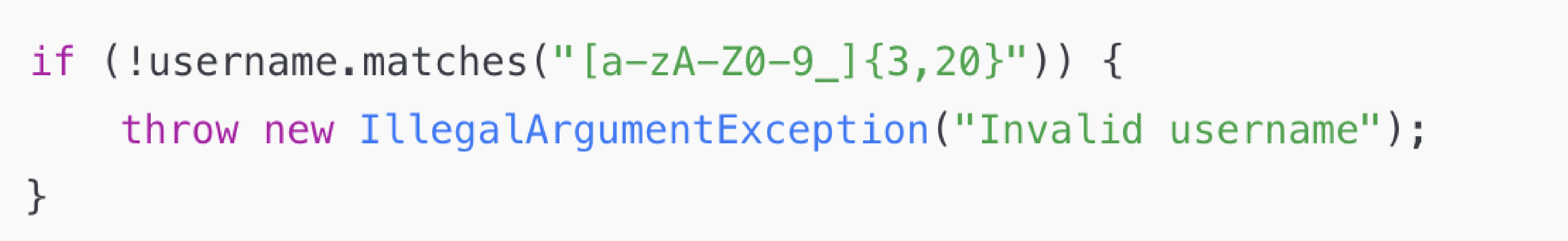
# Prevention Strategies

- **Core Principle: Never Trust User Input**
  - Validate and sanitize input
  - Client-side validation is not enough
  - Always validate on the server
  - Use whitelist patterns

# Prevention Strategies

## ❖ Core Principle: Never Trust User Input

- Validate and sanitize input
- Client-side validation is not enough
- Always validate on the server
- Use whitelist patterns



```
if (!username.matches("[a-zA-Z0-9_]{3,20}")) {
    throw new IllegalArgumentException("Invalid username");
}
```

# Use Prepared Statements

- **Parameterized queries as the gold standard**

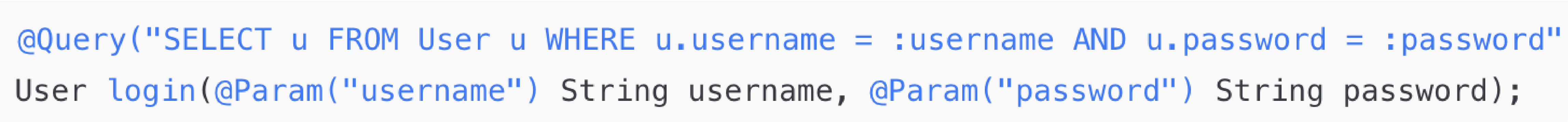
```
String query = "SELECT * FROM users WHERE username = ? AND password = ?";
return jdbcTemplate.queryForObject(query, new Object[]{username, password}, new
UserRowMapper());
```

- Prevents SQL from being altered
- Parameters are bound, not concatenated

# Spring Data JPA Best Practices

❖ **Let JPA handle query parameters**

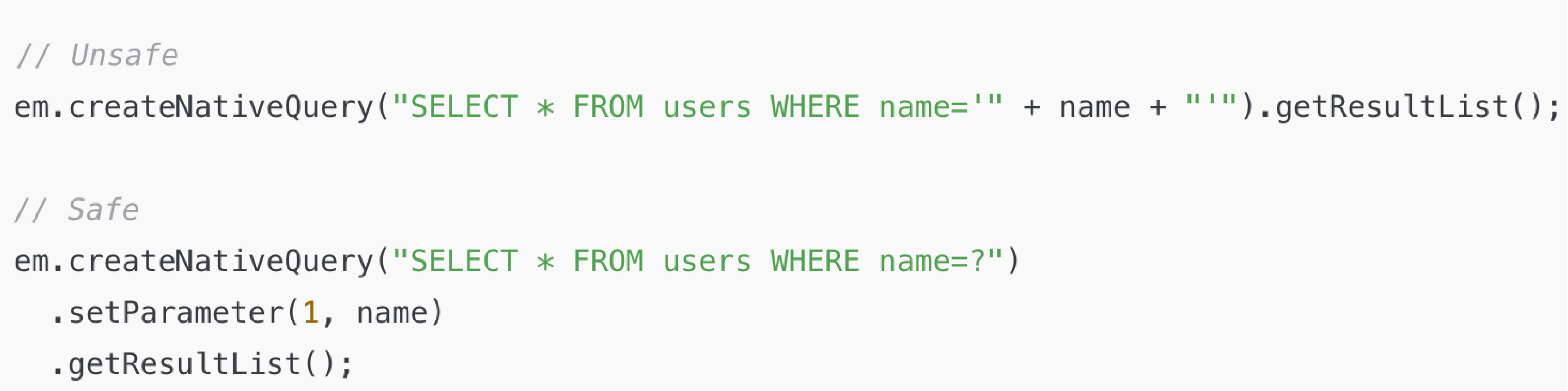
- ➢ Avoid string concatenation in JPQL
- ➢ Use @Param to bind parameters



```
@Query("SELECT u FROM User u WHERE u.username = :username AND u.password = :password"
User login(@Param("username") String username, @Param("password") String password);
```

# Avoid Native Queries (If Possible)

- **Native queries increase injection risk**
  - If you must use them, parameterize carefully
  - Example with **EntityManager** safe usage



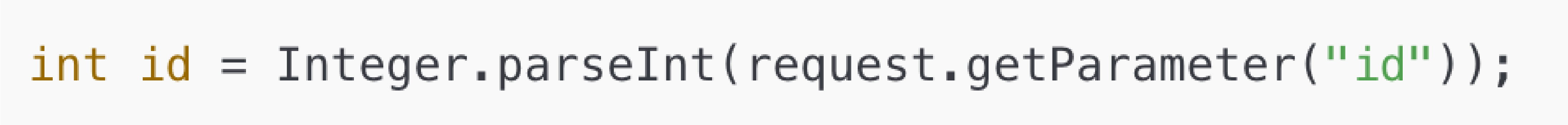
```
// Unsafe
em.createNativeQuery("SELECT * FROM users WHERE name='" + name + "'").getResultList();

// Safe
em.createNativeQuery("SELECT * FROM users WHERE name=?")
  .setParameter(1, name)
  .getResultList();
```

# Input Validation & Encoding

❖ **Defensive coding layers**

- Use regex for allowed characters
- Encode output to avoid stored XSS in combination attacks

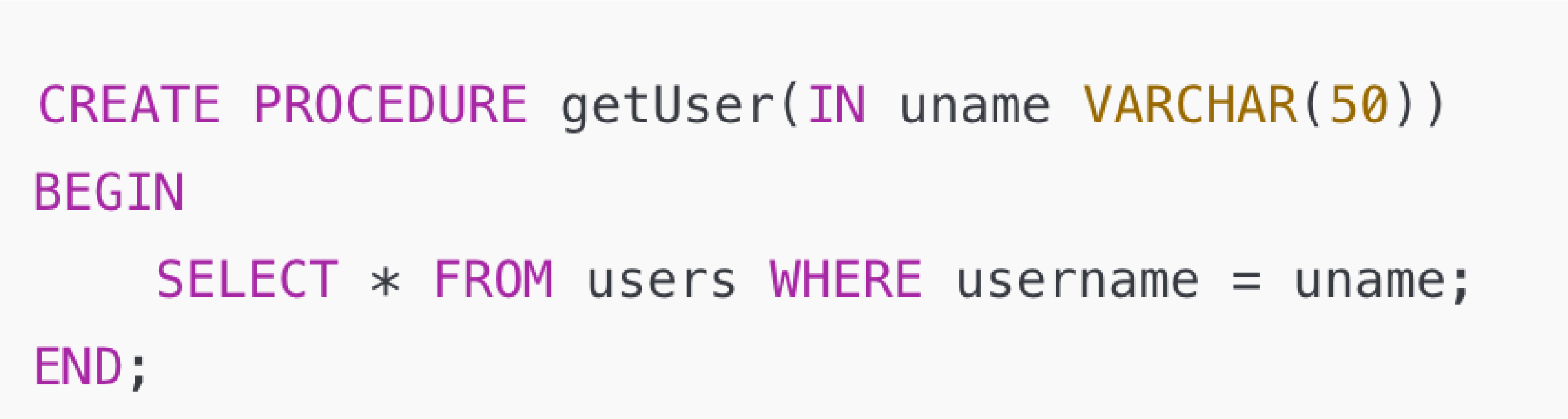


```
int id = Integer.parseInt(request.getParameter("id"));
```

# Stored Procedures & ORM Safety

## ❖ Additional mitigation layers

- ➢ Stored procedures can limit direct SQL access
- ➢ ORM frameworks reduce dynamic query building
- ➢ Called from Java with parameter binding



```
CREATE PROCEDURE getUser(IN uname VARCHAR(50))
BEGIN
    SELECT * FROM users WHERE username = uname;
END;
```

# Security Testing Tools

- **Detect vulnerabilities early**
  - Static code analysis (SonarQube, Checkmarx)
  - Dynamic testing (OWASP ZAP, Burp Suite)