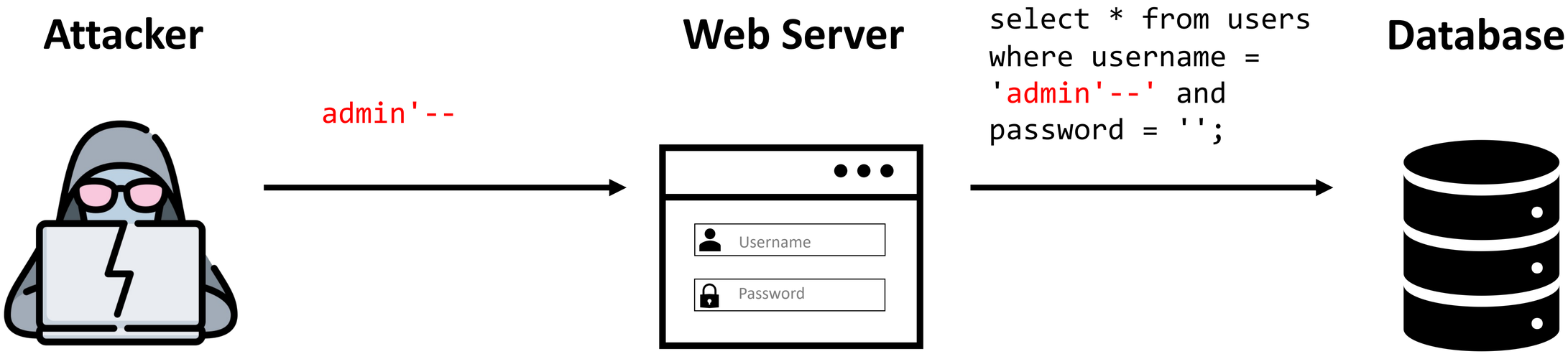
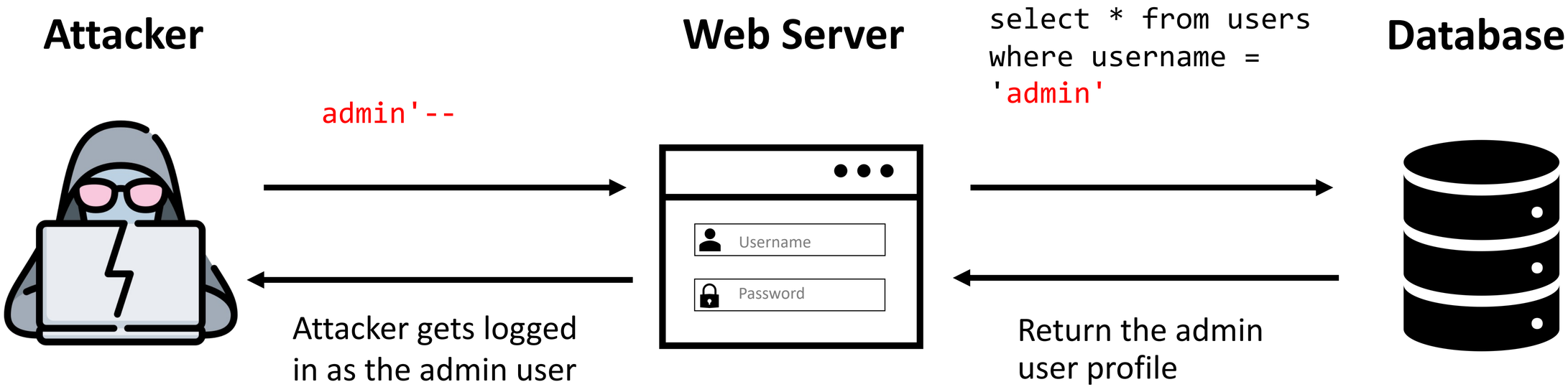
SQL Injection

- Vulnerability that consists of an attacker interfering with the SQL queries that an application makes to a database.



SQL Injection

- Vulnerability that consists of an attacker interfering with the SQL queries that an application makes to a database.



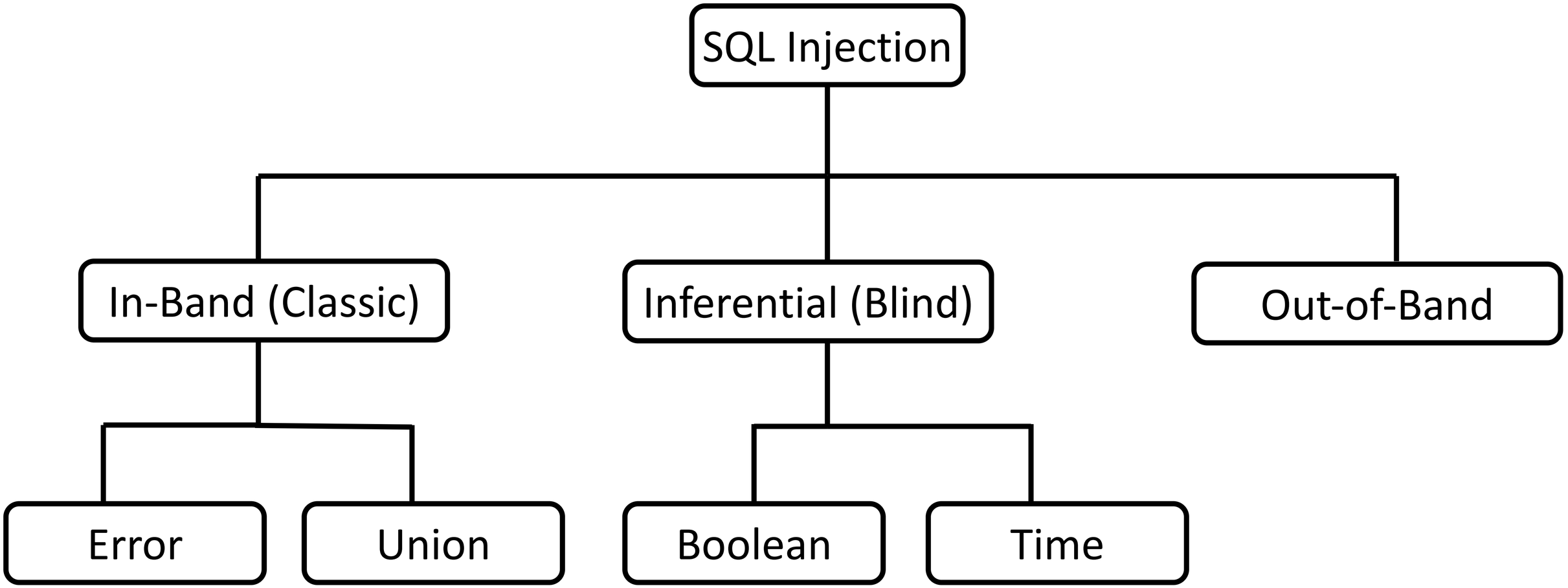
OWASP Top 10

OWASP Top 10 - 2013	OWASP Top 10 - 2017	OWASP Top 10 - 2021
A1 – Injection	A1 – Injection	A1 – Broken Access Control
A2 – Broken Authentication and Session Management	A2 – Broken Authentication	A2 – Cryptographic Failures
A3 – Cross-Site Scripting (XSS)	A3 – Sensitive Data Exposure	A3 - Injection
A4 – Insecure Direct Object References	A4 – XML External Entities (XXE)	A4 – Insecure Design
A5 – Security Misconfiguration	A5 – Broken Access Control	A5 – Security Misconfiguration
A6 – Sensitive Data Exposure	A6 – Security Misconfiguration	A6 – Vulnerable and Outdated Components
A7 – Missing Function Level Access Control	A7 – Cross-Site Scripting (XSS)	A7 – Identification and Authentication Failures
A8 – Cross-Site Request Forgery (CSRF)	A8 – Insecure Deserialization	A8 – Software and Data Integrity Failures
A9 – Using Components with Known Vulnerabilities	A9 – Using Components with Known Vulnerabilities	A9 – Security Logging and Monitoring Failures
A10 – Unvalidated Redirects and Forwards	A10 – Insufficient Logging & Monitoring	A10 – Server-Side Request Forgery (SSRF)

LAB EXERCISE #1

SQL injection vulnerability allowing login bypass

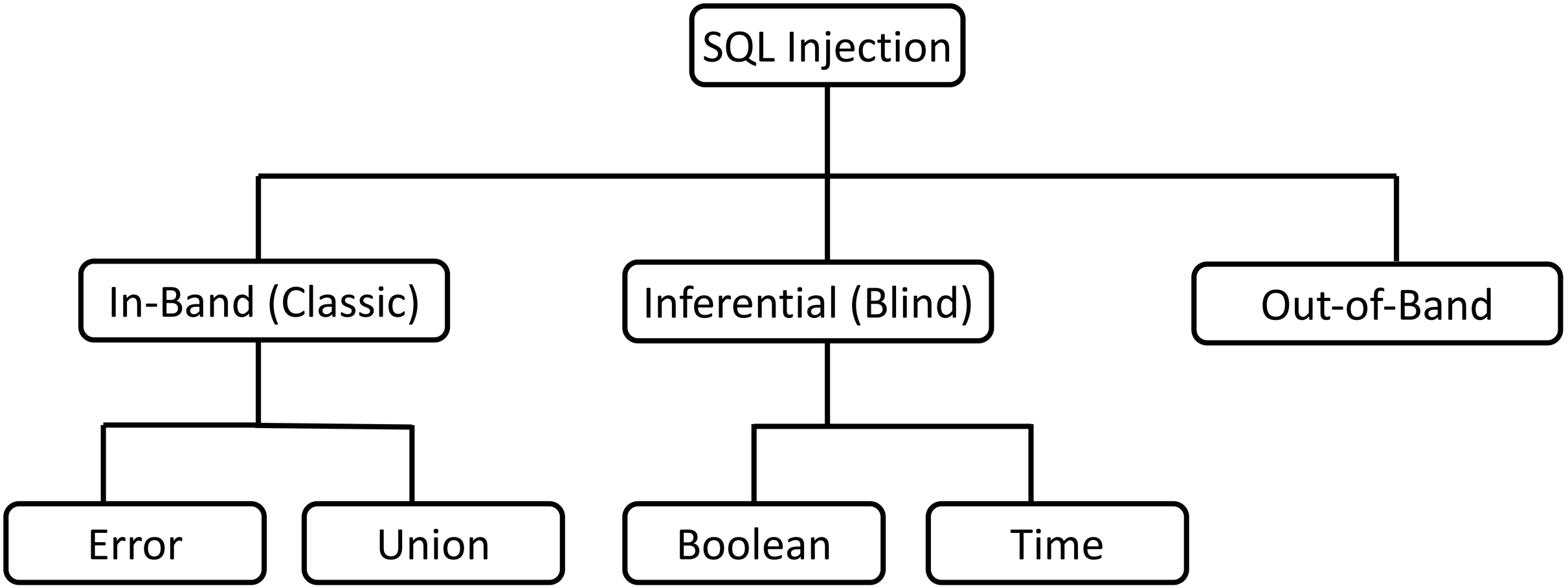
Types of SQL Injection



LAB EXERCISE #2

SQL injection UNION attack, retrieving data from other tables

Types of SQL Injection



LAB EXERCISE #3

Blind SQL injection with conditional responses

Preventing SQLi Vulnerabilities

- Primary Defenses:
  - **Use of Prepared Statements (Parameterized Queries)**

- Additional Defenses:
  - Also: Enforcing Least Privilege
  - Also: Performing Whitelist Input Validation as a Secondary Defense

Use of Prepared Statements

Code vulnerable to SQLi:

```
String query = "SELECT account_balance FROM user_data WHERE user_name = "
             + request.getParameter("customerName");
try {
    Statement statement = connection.createStatement( ... );
    ResultSet results = statement.executeQuery( query );
}
...
```

Spot the issue?

- User supplied input “cutomerName” is embedded directly into the SQL statement

Use of Prepared Statements

The construction of the SQL statement is performed in two steps:

- The application specifies the query's structure with placeholders for each user input
- The application specifies the content of each placeholder

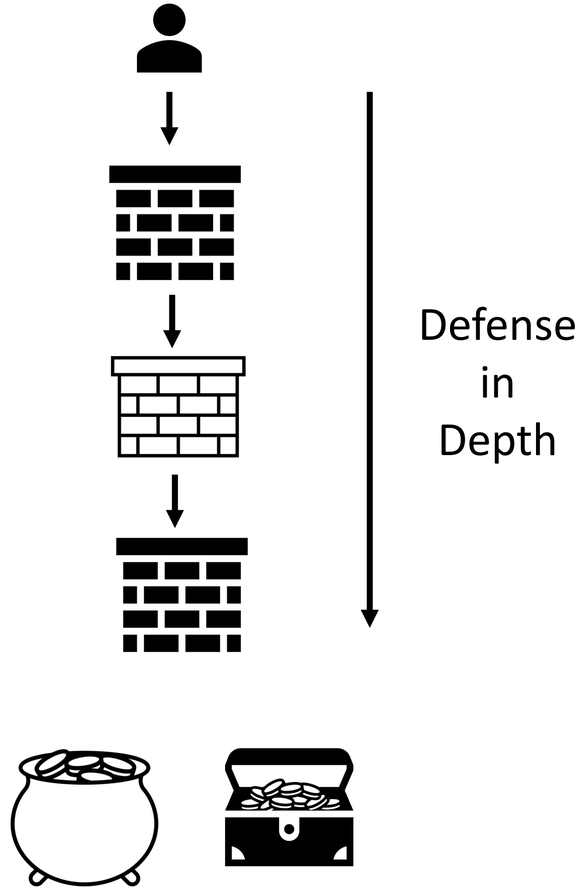
Code not vulnerable to SQLi:

```
// This should REALLY be validated too
String custname = request.getParameter("customerName");
// Perform input validation to detect attacks
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery( );
```

Additional Defenses

Least Privilege

- The application should use the lowest possible level of privileges when accessing the database
- Any unnecessary default functionality in the database should be removed or disabled
- Ensure CIS benchmark for the database in use is applied
- All vendor-issued security patches should be applied in a timely fashion



Allow List Input Validation

- Only accept characters included in the allow list.