**CASE STUDY 1: DETECTING SQL INJECTION AND CROSS-SITE SCRIPTING (XSS) ATTACKS IN WEB APPLICATIONS**

**Introduction**

Web applications are the backbone of modern digital services, providing platforms for e-commerce, banking, healthcare, and various other domains. However, these applications are often targeted by cybercriminals who exploit vulnerabilities to gain unauthorized access to sensitive data. Among the most prevalent and dangerous attacks are **SQL Injection (SQLi) and Cross-Site Scripting (XSS)** attacks.

This case study focuses on setting up a detection mechanism for both these attacks in incoming requests, preventing malicious exploitation of web applications.

**OVERVIEW OF ATTACKS**

**SQL Injection (SQLi)**

SQL Injection is a code injection attack that allows attackers to manipulate SQL queries executed by a web application. By injecting malicious SQL statements, attackers can bypass authentication, access confidential information, delete or modify database contents, and even gain administrative control over the system.

**Example of an SQL Injection Attack**

Consider a login form where users enter their credentials:

<form method="POST" action="login.php">

<input type="text" name="username" placeholder="Username">

<input type="password" name="password" placeholder="Password">

<input type="submit" value="Login">

</form>

A vulnerable backend query might be:

SELECT \* FROM users WHERE username = '$username' AND password = '$password';

An attacker could input the following in the username field:

admin' OR '1'='1' --

Which modifies the query to:

SELECT \* FROM users WHERE username = 'admin' OR '1'='1' --' AND password = '';

Since '1'='1' always evaluates to true, the attacker gains unauthorized access.

**Cross-Site Scripting (XSS)**

XSS attacks inject malicious scripts into web pages, which are then executed in the victim’s browser. This attack is used to steal cookies, session tokens, or redirect users to malicious websites.

**Example of an XSS Attack**

A vulnerable comment submission form:

<form method="POST" action="comment.php">

<textarea name="comment"></textarea>

<input type="submit" value="Post Comment">

</form>

If user input is not sanitized, an attacker could inject:

<script>alert('Your session has been hijacked!');</script>

This script executes when another user views the comment.

**SETTING UP DETECTION MECHANISMS**

To detect SQL Injection and XSS attacks in incoming requests, we need a multi-layered approach:

**Input Validation**

Strict input validation should be enforced to ensure only expected values are accepted. Some key validation techniques include:

* Allowlist approach: Accept only predefined characters.
* Reject special characters such as ', ", <, >, ;, --, #, /\* \*/, etc.
* Use parameterized queries to prevent SQLi.

**Example of Parameterized Queries in PHP**

$stmt = $conn->prepare("SELECT \* FROM users WHERE username = ? AND password = ?");

$stmt->bind\_param("ss", $username, $password);

$stmt->execute();

This prevents SQL injection as user input is treated as data, not SQL commands.

**Escaping Special Characters**

For XSS prevention, special characters should be encoded to prevent script execution.

**Example: Encoding User Input in PHP**

$comment = htmlspecialchars($\_POST['comment'], ENT\_QUOTES, 'UTF-8');

This ensures <script> is stored and displayed as &lt;script&gt;, preventing execution.

**Web Application Firewall (WAF)**

A WAF can monitor HTTP requests for malicious patterns and block attacks before they reach the application.

**Regular Expression-Based Detection**

A lightweight solution involves checking for SQLi and XSS attack patterns using regex.

**Example: Basic SQLi and XSS Detection in PHP**

function detectInjection($input) {

$pattern = "/(\bSELECT\b|\bUNION\b|\bINSERT\b|\bUPDATE\b|\bDELETE\b|\bDROP\b|\b--\b|['\";])/i";

return preg\_match($pattern, $input);

}

function detectXSS($input) {

$pattern = "/(<script|alert\(|onerror|onload|javascript:|<iframe)/i";

return preg\_match($pattern, $input);

}

$user\_input = $\_POST['data'];

if (detectInjection($user\_input) || detectXSS($user\_input)) {

die("Malicious input detected!");

}

This script blocks requests containing suspicious SQL or JavaScript patterns.

**Logging and Monitoring**

Implement logging for suspicious requests and analyze patterns over time.

* Use intrusion detection systems (IDS) like **Snort** or **OSSEC**.
* Store logs of rejected inputs for forensic analysis.

**TESTING THE DETECTION SYSTEM**

To validate the setup, we simulate SQLi and XSS attacks.

**SQL Injection Test Cases**

|  |  |
| --- | --- |
| **Test Input** | **Expected Response** |
| ' OR '1'='1' -- | Blocked |
| UNION SELECT null, username, password FROM users | Blocked |
| DROP TABLE users; -- | Blocked |

**XSS Test Cases**

|  |  |
| --- | --- |
| **Test Input** | **Expected Response** |
| <script>alert('Hacked');</script> | Blocked |
| <img src=x onerror=alert('XSS')> | Blocked |
| javascript:alert('Attack!') | Blocked |

**CONCLUSION**

SQL Injection and Cross-Site Scripting attacks remain significant threats to web security. By implementing **input validation, escaping special characters, using parameterized queries, deploying a WAF, and monitoring logs**, we can effectively detect and prevent such attacks. This case study provides a foundation for securing web applications and mitigating risks associated with malicious user inputs.