ISO 27001 Compliant Incident Management

SQL Injection Vulnerability Report

Introduction

This report outlines the identification and exploitation of a specific SQL injection vulnerability in the Damn Vulnerable Web Application (DVWA). Conducted in a controlled environment, the assessment illustrates how such vulnerabilities can significantly compromise application security and data integrity.

Incident Description

During the security assessment of DVWA, an SQL injection vulnerability was identified within the “SQL Injection” module. This flaw allows an attacker to manipulate SQL queries by injecting malicious code into input fields, leading to unauthorized access and data manipulation.

Reproduction Process

To demonstrate the vulnerability, the following SQL payload was used in the “User ID” field:

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

Explanation of the Payload

- **`1' OR '1'='1`**: This payload alters the logic of the SQL query to always return true, effectively bypassing any authentication checks.
- **Resulting Behavior**: Instead of retrieving a single user, the modified query retrieves all records from the database, exposing sensitive information.

The output from executing this injection returned:

- **ID:** 1’ OR ‘1’=’1
  First Name: admin
  Surname: admin
- **ID:** 1’ OR ‘1’=’1
  First Name: Gordon
  Surname: Brown
- **ID:** 1’ OR ‘1’=’1
  First Name: Hack
  Surname: Me
- **ID:** 1’ OR ‘1’=’1
  First Name: Pablo
  Surname: Picasso
- **ID:** 1’ OR ‘1’=’1
  First Name: Bob
  Surname: Smith

This successful injection demonstrates how an attacker can gain unauthorized access to all user data stored in the application.

Incident Impact

Exploiting this vulnerability can lead to:

- **Data Exposure**: Unauthorized access to confidential user information, including usernames and passwords.
- **Data Manipulation**: The potential to alter, delete, or compromise sensitive data.
- **Loss of Trust**: Erosion of user trust in the application and the organization’s ability to protect sensitive data.
- **Regulatory Consequences**: Non-compliance with data protection regulations, which can result in penalties.

Recommendations

1. Input Validation

Implement rigorous validation to ensure that user inputs conform to expected formats.

Example in PHP:

```
function validateUserId($userId) {
    // Ensure the ID is a positive integer
    if (!filter_var($userId, FILTER_VALIDATE_INT) || $userId
        <= 0) {
        throw new InvalidArgumentException('Invalid ID.');
    }
    return $userId;
}

// Usage
$userId = validateUserId($_POST['user_id']);
```

2. Prepared Statements

Use prepared statements and parameterized queries to separate SQL code from user data.

Example in PHP using PDO:

```
try {
    $pdo = new PDO('mysql:host=localhost;dbname=my_database',
        'username', 'password');
    $pdo->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);

    // Prepare the query
    $stmt = $pdo->prepare('SELECT * FROM users WHERE id
        = :id');
    $stmt->bindParam(':id', $userId, PDO::PARAM_INT);

    // Execute the query
```

```
    $stmt->execute();

    // Fetch the results
    $results = $stmt->fetchAll(PDO::FETCH_ASSOC);
} catch (PDOException $e) {
    echo 'Error: ' . $e->getMessage();
}
```

3. Web Application Firewall (WAF)

Implement a web application firewall to monitor and filter HTTP traffic.

Basic WAF Configuration:

```
# Example rules configuration for a WAF like ModSecurity
SecRuleEngine On
SecRequestBodyAccess On
SecRule ARGS ".*([';]+).*" "id:1001,phase:
        2,t:none,log,deny,status:403"
```

4. Regular Security Audits

Conduct regular security audits to identify and remediate vulnerabilities.

Tool Example: - **Using OWASP ZAP**: Run a vulnerability scan on your application.

```
zap.sh -quickurl http://your_application -quickout
        report.html
```

5. User Education

Provide training to developers on secure coding practices.

Training Content: - Importance of input validation. - How to implement prepared statements. - Recognizing common attack patterns.

6. Database Security

Apply the principle of least privilege to database accounts.

SQL Example:

```sql
-- Create a user with limited permissions
CREATE USER 'new_user'@'localhost' IDENTIFIED BY 'password';
GRANT SELECT, INSERT ON database_name.* TO
        'new_user'@'localhost';
```

7. Error Handling

Implement generic error messages that do not reveal details about the database or application.

Example in PHP:

```php
try {
    // Code that may throw exceptions
} catch (Exception $e) {
    // Display a generic error message
    echo 'An error occurred. Please try again later.';
}
```

By adopting these recommendations, organizations can significantly enhance their defenses against SQL injection vulnerabilities, protecting sensitive data and maintaining the integrity of their applications.

Conclusion

The SQL injection vulnerability identified in the Damn Vulnerable Web Application (DVWA) highlights critical security risks in web applications. To mitigate these risks, organizations should implement the following measures:

1. **Input Validation**: Ensure user inputs are rigorously validated to match expected formats.
2. **Prepared Statements**: Use prepared statements and parameterized queries to separate SQL code from user data, preventing injection.
3. **Web Application Firewall (WAF)**: Deploy a WAF to monitor and filter incoming HTTP traffic for malicious patterns.

4. **Regular Security Audits**: Conduct frequent security assessments and vulnerability scans to identify and address potential weaknesses.
5. **User Education**: Train developers on secure coding practices to reduce the risk of vulnerabilities.
6. **Database Security**: Apply the principle of least privilege to limit database access and permissions.
7. **Error Handling**: Implement generic error messages that do not disclose sensitive information about the application or database.

By following these recommendations, organizations can significantly enhance their security posture and protect sensitive data from potential attacks.