OWASP Top 10 Vulnerabilities: Injection and Insecure Design

Daniela Tomás
up202004946@edu.fc.up.pt

Diogo Nunes
up202007895@edu.fc.up.pt

João Veloso
up202005801@edu.fc.up.pt

***Abstract*—This document analyzes injection and insecure design vulnerabilities in two distinct websites: an insecure and a secure website, discussing particular examples, potential risks, and countermeasures to mitigate against these threats.**

***Index Terms*—cybersecurity, injection, insecure design, exploitation, countermeasures**

I. Introduction

This document provides a thorough analysis of two different websites: one that highlights unsafe behaviors vulnerable to various types of cyberattacks, and another that shows safe precautions to prevent these vulnerabilities. The main goal of the analysis is to address the major issues with insecure design [1] and injection [2] that are mentioned in the OWASP.

By analyzing critical areas such as weak password encryption, lack of input sanitization, potential SQL injection threats, and command injection vulnerabilities, we can show the importance of robust security measures in web applications.

Through a comparative analysis between insecure and secure practices, this report emphasizes the importance of using secure coding practices to protect web applications against potential cyberattacks.

II. A03:2021 - Injection

A. Exploitation Techniques and Examples

1) Lack of Sanitization in Login: The absence of input sanitization in a login system can lead to various security vulnerabilities, particularly exposing the application to attacks like SQL injection, where attackers insert malicious SQL statements into input fields in order to manipulate the database.

```
$query = '
   SELECT customerId, email
   FROM Customer
   WHERE lower(email) = "' .
       strtolower($email) . '" AND password =
       "' . sha1($password) . '"
';

$stmt = $db->query($query);
```

This code is vulnerable to SQL injection due to the direct concatenation of variables into the SQL query string. If `$email` contains malicious input like `alice@example.com" --` (alice@example.com is a valid email), an attacker can manipulate the query by cutting out the part of the query that utilizes the password and have unauthorized access to the system [3].

Note: SQLite in PHP doesn't support multiple queries in a single execution by default for security reasons. Therefore, on our insecure website, it is not possible to perform, for example, a DROP TABLE Customer after the SELECT.

2) Lack of Sanitization in Shell Commands: Lack of sanitization in shell commands may expose the system to potential command injections.

```
if (!empty($hostname)) {
   $pingResult = shell_exec("ping -c 4 " .
       $hostname);
   echo "<pre>Ping results for
       $hostname:\n$pingResult</pre>";
}
```

This script checks the connectivity to a given hostname or IP address using `shell_exec` to execute the ping command and then displays the ping results.

It doesn't properly validate or sanitize the input. Therefore, if we provide, for example, `8.8.8.8; ls -la` as input, the ls command will be executed after the ping [4].

B. Countermeasures

1) Use of Prepared Statements (Parameterized Queries): Prepared statements ensure separation between SQL code and user input, mitigating SQL injection risks.

```
$stmt = $db->prepare('
   SELECT customerId, email
   FROM Customer
   WHERE lower(email) = ?
');

$stmt->execute(array(strtolower($email)));
```

2) Use of Stored Procedures: Utilizing stored procedures further protects against SQL injection by encapsulating SQL logic within the database.

3) Escaping all User Supplied Input: String escaping or input validation helps prevent unintended SQL commands from being executed.

4) Sanitized Hostname in Shell Commands: Implementing sanitization in shell commands prevents command injections.

```
if (!empty($hostname)) {
   $sanitizedHostname =
       escapeshellcmd($hostname);
```

```
    $pingResult = shell_exec("ping -c 4 " .
        $sanitizedHostname);

    echo "<pre>Ping results for
        $sanitizedHostname:\n$pingResult</pre>";
}
```

It is a good practice to use `escapeshellcmd` to prevent shell injection, as it safely uses user input within shell commands by escaping characters that have particular meaning in the command shell. For individual arguments inside shell commands, `escapeshellarg` can also be used.

III. A04:2021 - Insecure Design

A. Exploitation Techniques and Examples

1) Weak Password Encryption (SHA1): In PHP, the `sha1` function generates a SHA-1 hash of a string.

However, the usage of SHA1 for password encryption is considered weak due to its susceptibility to brute-force and rainbow table attacks, making it relatively easy for attackers to crack hashed passwords [5].

2) Detailed Error Messages: Detailed error messages expose potential vulnerabilities to attackers by providing too much information about the system's inner workings [6].

```
if(!$customer) {
    $session->addMessage('error', "Invalid
        email " . $_POST['email']);
}
...
  else {
      $session->addMessage('error',"Invalid
          password " . $_POST['password']);
  }
```

Revealing information like ”Invalid email” or ”Invalid password” can be a privacy concern. Attackers could use this information to get hints and identify valid user accounts.

3) No Maximum Quantity Limited to Stock: Lack of enforcement for maximum quantity could lead to inventory manipulation or depletion.

```
<p>Quantity: <input name="quantity"
    type="number" value="1" min="1"
    step="0"></p>
```

This code creates a numeric input field where users can select the quantity of a book they want to reserve. The code allows a user to enter quantities that exceed the available stock.

Large purchases or reservations by a single customer could deplete or even exceed the stock, leaving other customers unable to access the book they want or causing losses for the store.

B. Countermeasures

1) Stronger Password Hashing Functions: SHA-1 is no longer considered secure for password storage due to its vulnerability to brute-force attacks. More secure hashing functions like SHA256, SHA512, or bcrypt should be used. To keep user credentials safe, hashing algorithms must be updated frequently in accordance with the most recent security standards.

Generating and validating passwords:

- Prepend salt to the password and hash it using a standard cryptographic function like bcrypt.

- Store both salt and hash in the user's database record.

- Retrieve the user's salt and hash for password validation.

Hash and validate passwords in PHP:

```
string password_hash (string $pwd, integer
    $algo [, array $opts])

boolean password_verify (string $pwd, string
    $hash)
```

`password_hash`: Generates a hash with its own salt.

`password_verify`: Validates the hash against the password without requiring separate storage for salt or algorithm.

2) Reduced Detailed Error Messages: Minimizing error message details limits the exposure of system information to potential attackers.

```
...
else {
   $session->addMessage('error', "Invalid
       email or password.");
}
```

”Invalid email or password.” informs the user that there was an issue with the provided credentials without specifying whether it was the email or password that was incorrect.

3) Enforcement of (Maximum) Stock Limits: To prevent inventory manipulation, we need to set limits.

```
<p>Quantity: <input name="quantity"
    type="number" value="1" min="1" max="10"
    step="0"</p>
```

The code includes a `min` and `max` attribute that prevents a user from entering quantities that exceed the available stock.

Also, since the `max` attribute can be edited on the client-side to allow higher limits, the server should validate the quantity sent by the client.

IV. Conclusion

In conclusion, a comprehensive strategy that incorporates strong countermeasures like input validation, prepared statements, secure hashing functions, and system exposure reduction is necessary to tackle injection and insecure design flaws. Web applications can strengthen their security against possible cyberattacks by putting these precautions into place.

References

[1] https://owasp.org/Top10/A04_2021-Insecure_Design/
[2] https://owasp.org/Top10/A03_2021-Injection/
[3] The MITRE Corporation, Common Weakness Enumeration, “CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')”. https://cwe.mitre.org/data/definitions/89.html, 2023.

[4] The MITRE Corporation, Common Weakness Enumeration, “CWE-78: Improper Neutralization of Special Elements used in an OS Command (’OS Command Injection’)”. https://cwe.mitre.org/data/definitions/78.html, 2023.
[5] The MITRE Corporation, Common Weakness Enumeration, “CWE-257: Storing Passwords in a Recoverable Format”. https://cwe.mitre.org/data/definitions/257.html, 2023.
[6] The MITRE Corporation, Common Weakness Enumeration, “CWE-209: Generation of Error Message Containing Sensitive Information”. https://cwe.mitre.org/data/definitions/209.html, 2023.
[7] https://owasp.org/www-project-top-ten/
[8] https://sectigostore.com/blog/what-is-owasp-what-are-the-owasp-top-10-vulnerabilities/
[9] Dafydd Stuttard and Marcus Pinto, The Web Application Hacker’s Handbook, 2nd ed, 2011.