ST10092354

Lerato Kekana

APDS7311 ICE Task 1

Question 1

Application security involves a set of measures, practices, and technologies designed to safeguard software applications against vulnerabilities and threats that may compromise data, functionality, or user experience. It covers a wide array of techniques and protections aimed at deterring malicious attacks, unauthorized access, data breaches, and other security incidents throughout the entire lifecycle of an application—from its design and development phases to deployment and ongoing maintenance.

Question 2

The risk of untrusted data emerges when an application receives input from unreliable sources like user input, external systems, or third-party services, without adequately validating or sanitizing it. Attackers can exploit such untrusted data to manipulate the application, breach security measures, and gain access to sensitive information. This is one of the most prevalent and critical vulnerabilities in application security.

Key Threats from Untrusted Data

These include:

Injection Attacks

Untrusted data can facilitate the injection of malicious code or commands into an application. Common types of injection attacks include:

- **SQL Injection:** Attackers insert malicious SQL code into a query, allowing them to manipulate databases, access unauthorized data, or delete records.
- **Command Injection:** This attack involves executing arbitrary commands on the server by inserting harmful input into an application that interacts with system commands.
- **LDAP Injection:** Malicious LDAP queries are crafted to manipulate directory services, enabling unauthorized access.

Example: In an SQL injection attack, an attacker might input '; DROP TABLE users; -- into a login field, potentially deleting the entire user database if the application fails to sanitize the input properly.

Cross-Site Scripting (XSS)

XSS (Cross-Site Scripting) occurs when untrusted data is included in web pages without adequate validation, allowing attackers to inject scripts that execute in the victim's browser. This vulnerability can lead to session hijacking, website defacement, or redirection to malicious sites.

Example: If an attacker submits <script>alert('Hacked!');</script> through a comment form, and the website displays this comment without sanitization, the script would execute in the browser of every user who visits that page.

Cross-Site Request Forgery (CSRF)

Untrusted data can also enable CSRF (Cross-Site Request Forgery) attacks. In this scenario, an attacker deceives a user into submitting a malicious request—such as transferring funds—without their consent by embedding untrusted data in web forms or links.

Remote Code Execution (RCE)

When an application processes untrusted data without verifying its content, attackers can send payloads that exploit vulnerabilities, enabling them to execute arbitrary code on either the server or the client.

Deserialisation Attacks

Some applications deserialize data inputs without proper validation, allowing attackers to create malicious objects that can manipulate application behavior, potentially leading to remote code execution or privilege escalation.

Open Redirects & URL Manipulation

Attackers can manipulate untrusted data, such as URL parameters, to redirect users to malicious websites, potentially leading to phishing attacks or malware distribution.

Mitigating the Threat of Untrusted Data:

- **Input Validation:** Rigorously validate and sanitize all input data to ensure it conforms to expected formats (e.g., length, type).
- **Parameterised Queries:** Use prepared statements or parameterized queries for database interactions to prevent SQL Injection.
- **Escaping Data:** Escape user input when displaying it in the browser to thwart XSS attacks.
- **Content Security Policy (CSP):** Implement a CSP to restrict the types of content that the browser can execute.
- **Whitelisting:** Validate input against a whitelist of allowed values (e.g., known file extensions or characters).
- **Encoding Output:** Encode outputs that could be interpreted as HTML, URLs, or SQL to prevent them from being executed as code.
- **Security Frameworks:** Utilise frameworks and libraries that help enforce security best practices, such as OWASP's security recommendations.

Question 3

HTTP (HyperText Transfer Protocol) is the fundamental protocol that powers web traffic, enabling communication between web browsers (clients) and servers. It supports the exchange of resources like HTML pages, images, and multimedia content over the internet. Operating on a request-response model, HTTP forms the backbone of how web pages and applications work.

Question 4

Blacklist input validation is an anti-pattern that involves comparing input data against a list of known harmful values or patterns and rejecting them. This approach operates under the assumption that blocking specific inputs will prevent malicious actions within the application.

Issues:

Easily Bypassed: Attackers can manipulate malicious inputs to avoid detection by the blacklist. For instance, blocking <script> might stop one XSS attack, but an attacker could obfuscate or encode their input (e.g., using variations like <ScRipT> or encoding the string) to bypass the filter.

Limited Coverage: Creating an exhaustive list of all potentially harmful inputs is nearly impossible, leaving the application exposed to new or unaccounted threats.

Maintenance Complexity: Maintaining an up-to-date blacklist is challenging, particularly as new vulnerabilities and attack methods continue to evolve.

Best Practices:

Opt for whitelist input validation instead, where only explicitly defined and allowed input—such as specific data formats, lengths, and types—is accepted. This approach is more secure, as it permits only known, safe inputs rather than attempting to block every potential malicious entry

Lack of Parameterised SQL

Anti-pattern: The absence of parameterised SQL arises when developers build SQL queries by directly concatenating user inputs into the query string, commonly known as dynamic SQL. This method poses a significant risk of SQL injection attacks, allowing an attacker to compromise the query structure by inserting malicious input.

Issues:

SQL Injection Vulnerability: If user input isn't properly sanitized, attackers can inject SQL commands into the query, potentially gaining unauthorized access to the database, modifying data, or executing administrative operations. For example, if the query is constructed as:

sql

Copy code

SELECT * FROM users WHERE username = '" + userInput + "';"

An attacker could input admin'--, turning the query into:

sql

Copy code

SELECT * FROM users WHERE username = 'admin'--';

This would comment out the rest of the query, bypassing authentication.

Best Practice:

Use parameterized queries or prepared statements, where user input is passed as a parameter, and the SQL engine ensures that the input is treated strictly as data, not part of the query logic. For example:

sql

Copy code

SELECT * FROM users WHERE username = ?;

This approach makes it impossible for an attacker's input to alter the query structure.

3. Use of Weak or Incorrect Ciphers

Anti-pattern: Relying on weak or inappropriate cryptographic algorithms involves using outdated, insecure, or misconfigured encryption methods to safeguard data. Examples include utilizing MD5 or SHA-1 for hashing passwords or sensitive information, or employing ECB mode for block ciphers, which can lead to vulnerabilities in data protection.

Issues:

Weak Cryptography: Algorithms such as MD5 and SHA-1 exhibit known vulnerabilities, including susceptibility to collision attacks, which make it easier for attackers to breach encrypted or hashed data.

Improper Cipher Modes: Utilising insecure modes of operation, such as ECB (Electronic Codebook) for block ciphers, can create predictable encryption patterns, thereby increasing the risk of data compromise. In ECB, identical plaintext blocks are encrypted into identical ciphertext blocks, potentially exposing patterns within the encrypted

Misconfiguration: Using incorrect key lengths, like a 64-bit key instead of a 256-bit key, or failing to manage cryptographic keys effectively can undermine the security of otherwise robust encryption algorithms.

Best Practices:

Utilise modern and secure cryptographic algorithms, such as AES (Advanced Encryption Standard) for encryption and SHA-256 or SHA-3 for hashing.

Implement secure modes of operation, like CBC (Cipher Block Chaining) or GCM (Galois/Counter Mode), when using block ciphers.

Regularly review and update cryptographic practices to ensure compliance with current security standards.

Conclusion

These anti-patterns highlight poor security practices that elevate the risk of attacks and data compromise:

Blacklist Input Validation: This approach focuses on blocking harmful inputs but is susceptible to evasion. Instead, use whitelist validation for more comprehensive protection.

Lack of Parameterized SQL: This vulnerability exposes applications to SQL injection attacks. Always use parameterized queries to mitigate injection risks.

Use of Weak or Incorrect Ciphers: Relying on inadequate cryptographic methods compromises data protection. Always opt for modern, secure cryptographic algorithms and configurations

Question 5

The login workflow typically consists of several steps to ensure secure user authentication. Here's a concise overview of the nine steps involved:

1. **User Navigates to Login Page:**
The user accesses the login page and enters their credentials (username/email and password).
    - **Security Note:** Use HTTPS to secure data transmission.
2. **User Submits Login Information:**
The user submits the form by clicking "Login," sending their credentials to the server via an HTTP POST request.
    - **Security Note:** Encrypt sensitive data like passwords during transmission.
3. **Server Receives and Parses the Request:**
The server extracts the login credentials from the incoming request.
    - **Security Note:** Sanitize input to prevent injection attacks.
4. **Server Fetches User Data:**
The server queries the authentication database to retrieve the user's stored credentials (username/email and hashed password).
    - **Security Note:** Ensure passwords are hashed and salted in the database.
5. **Password Hashing and Comparison:**
The server hashes the submitted password and compares it to the stored hash.
    - **Security Note:** Always compare hashed values, never raw passwords.
6. **Authentication Decision:**
If the hashes match, login is successful; otherwise, it fails, and the user is notified.

- **Security Note:** Limit failed login attempts to prevent brute-force attacks.

7. **Session or Token Generation:**

Upon successful authentication, the server generates a session ID or token to identify the user in future requests.

- **Security Note:** Securely sign tokens to prevent tampering.

8. **Session or Token Sent to Client:**

The server sends the session ID or token back to the client via an HTTP cookie or response body.

- **Security Note:** Mark cookies as HttpOnly and Secure to prevent theft.

9. **Client Stores Session or Token:**

The client stores the session ID or token, typically in a cookie or local/session storage.

- **Security Note:** Avoid storing sensitive tokens in localStorage to reduce XSS vulnerability.

This streamlined process ensures secure user authentication while implementing best security practices at each step.