**APDS7311**

**ICE TASK 1**

Question 1

Application security refers to the measures, practices, and technologies put in place to protect software applications from vulnerabilities and threats that could compromise data, functionality, or user experience. It encompasses a broad range of techniques and safeguards aimed at preventing malicious attacks, unauthorized access, data breaches, and other security incidents throughout an application’s lifecycle—from design and development to deployment and maintenance.

Question 2

The threat of untrusted data arises when an application receives input from an unreliable source, such as user input, external systems, or third-party services, and fails to properly validate or sanitize it. Untrusted data can be exploited by attackers to manipulate the application, compromise security, and access sensitive information. This is one of the most common and dangerous vulnerabilities in application security.

Key threats from untrusted data include:

1. Injection Attacks

Untrusted data can be used to inject malicious code or commands into an application. Common types of injection attacks include:

SQL Injection: Malicious SQL code is inserted into a query, allowing attackers to manipulate databases, access unauthorized data, or delete records.

Command Injection: Attackers execute arbitrary commands on the server by inserting malicious input into an application that passes data to system commands.

LDAP Injection: Malicious LDAP queries are crafted to manipulate directory services and gain unauthorized access.

Example: In SQL Injection, an attacker could input '; DROP TABLE users; -- in a login field, potentially leading to the deletion of the entire user database if the application doesn't sanitize the input.

2. Cross-Site Scripting (XSS)

XSS occurs when untrusted data is included in web pages without proper validation, allowing attackers to inject scripts that run in the victim’s browser. This can lead to session hijacking, defacement, or redirection to malicious sites.

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

3. Cross-Site Request Forgery (CSRF)

Untrusted data can also facilitate CSRF attacks. An attacker tricks a user into submitting a malicious request, such as transferring funds, without their consent, by embedding untrusted data in web forms or links.

4. Remote Code Execution (RCE)

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

5. Deserialization Attacks

Some applications deserialize data inputs without validation, which could allow attackers to craft malicious objects and manipulate application behavior, leading to remote code execution or privilege escalation.

6. Open Redirects and URL Manipulation

Attackers can manipulate untrusted data such as URL parameters to redirect users to malicious websites, which could result in phishing attacks or malware distribution.

Mitigating the Threat of Untrusted Data:

Input Validation: Rigorously validate and sanitize all input data, ensuring it matches the expected format (e.g., length, type).

Parameterized 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 prevent XSS attacks.

Content Security Policy (CSP): Implement a CSP to limit the types of content the browser can execute.

Whitelisting: Validate input against a whitelist of allowed values (e.g., known file extensions, characters).

Encoding Output: Encode outputs that could be used in HTML, URLs, or SQL to avoid interpreting them as executable code.

Security Frameworks: Utilize frameworks and libraries that help enforce security practices, like OWASP's security recommendations.

Question 3

HTTP (HyperText Transfer Protocol) is the foundational protocol that drives the flow of web traffic by enabling communication between web browsers (clients) and web servers. It facilitates the exchange of resources such as HTML pages, images, and multimedia content over the internet. HTTP operates on a request-response model, which is the backbone of how web pages and applications function.

Question 4

Blacklist Input Validation

Anti-pattern: Blacklist input validation involves checking input data against a list of known bad values (or patterns) and rejecting them. This method assumes that by blocking certain inputs, the application can prevent malicious activity.

Issues:

Easily Bypassed: Attackers can use variations of malicious inputs that aren't on the blacklist. For example, blocking <script> may prevent one XSS attack, but an attacker can obfuscate or encode their input to bypass this check (e.g., using a variation like <ScRipT> or encoding the string).

Limited Coverage: It’s difficult to create a comprehensive list of all potentially dangerous inputs, leaving gaps in security.

Maintenance Complexity: Keeping the blacklist updated is challenging, especially as new vulnerabilities and attack techniques emerge.

Best Practice:

Use whitelist input validation instead, where you explicitly define what is allowed (e.g., data formats, length, types). This is more secure, as it only allows known, safe input rather than trying to block every potential malicious input.

2. Lack of Parameterized SQL

Anti-pattern: The lack of parameterized SQL occurs when developers construct SQL queries by concatenating user inputs directly into the query string, often referred to as dynamic SQL. This practice is highly vulnerable to SQL injection attacks, where an attacker can manipulate query structure by injecting 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: Using weak or incorrect cryptographic algorithms or ciphers means employing outdated, insecure, or improperly configured encryption methods to protect data. Examples include using MD5 or SHA-1 for hashing passwords or sensitive data, or using ECB mode for block ciphers.

Issues:

Weak Cryptography: Algorithms like MD5 and SHA-1 have known vulnerabilities (e.g., susceptibility to collision attacks), making it easier for attackers to crack encrypted or hashed data.

Improper Cipher Modes: Using insecure modes of operation like ECB (Electronic Codebook) for block ciphers can lead to predictable encryption patterns, making the data easier to compromise. ECB encrypts identical plaintext blocks into identical ciphertext blocks, which can reveal patterns in the encrypted data.

Misconfiguration: Incorrect key lengths (e.g., using a 64-bit key instead of a 256-bit key) or poor management of cryptographic keys can weaken the security of otherwise strong encryption algorithms.

Best Practice:

Use modern, 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) for block ciphers.

Regularly review and update cryptographic practices to ensure they meet current security standards.

Conclusion

These anti-patterns represent poor security practices that increase the risk of attack and data compromise:

Blacklist Input Validation focuses on blocking bad inputs, but is prone to bypasses; use whitelist validation for more comprehensive protection.

Lack of Parameterized SQL opens the door to SQL injection attacks; use parameterized queries to prevent injection vulnerabilities.

Use of Weak or Incorrect Ciphers leads to inadequate data protection; always use modern, secure cryptographic methods and configurations.

Question 5

The login workflow typically involves a sequence of steps that ensures the secure authentication of a user. Here's a breakdown of the nine steps of a typical login workflow:

1. User Navigates to Login Page

The user accesses the login page of the application. This page provides input fields for the user to enter their credentials (usually a username or email and password).

Client-side actions: User inputs their credentials into a form.

Security note: The page should use HTTPS to secure the transmission of data.

2. User Submits Login Information

Once the user inputs their credentials, they submit the form by clicking a "Login" or "Submit" button. This triggers a request from the client (browser, mobile app, etc.) to the server containing the user's login credentials.

Client-side action: The credentials are sent as part of an HTTP POST request to the server.

Security note: Sensitive data like passwords should be encrypted during transmission.

3. Server Receives and Parses the Request

The server receives the user's request and extracts the login credentials (username and password) from the payload.

Server-side action: The server processes the incoming data and prepares to verify the credentials.

Security note: Input should be sanitized to prevent injection attacks like SQL injection.

4. Server Fetches User Data

The server queries its authentication database or identity provider to retrieve the user’s stored credentials (typically the username or email and a hashed password).

Server-side action: The server searches the database for the matching user record based on the submitted username or email.

Security note: Ensure the password is hashed and salted in the database.

5. Password Hashing and Comparison

Once the server retrieves the stored (hashed) password for the user, it hashes the password provided by the user in the login form using the same hashing algorithm and salt that was used to store the password. Then, it compares the resulting hash with the stored hash.

Server-side action: The server applies the same cryptographic hashing function (e.g., bcrypt, Argon2) to the submitted password and compares it to the stored hash.

Security note: Never store or compare raw passwords; always compare hashed values.

6. Authentication Decision

If the hashed password matches the stored hashed password, the server considers the login successful. If not, the authentication fails, and the user is notified.

Server-side action: A decision is made (either accept or reject).

Security note: Limit the number of failed login attempts to prevent brute-force attacks.

7. Session or Token Generation

If authentication is successful, the server generates a session or token to represent the authenticated user. This session/token is used to identify the user in subsequent requests without requiring them to re-enter their credentials.

Server-side action: The server creates a session ID (for session-based authentication) or a JSON Web Token (JWT) (for token-based authentication).

Security note: The token should be securely signed and contain the user's identity and expiration time to prevent tampering.

8. Session or Token Sent to Client

The server sends the session ID or token back to the client, typically through an HTTP cookie (for session IDs) or in the response body (for tokens).

Server-side action: The token is sent securely to the client.

Security note: Cookies containing session IDs should be marked as HttpOnly and Secure to prevent theft via JavaScript or insecure channels.

9. Client Stores Session or Token

The client receives the session ID or token and stores it. For session-based authentication, it will usually be stored in a cookie. For token-based authentication (e.g., JWT), it may be stored in localStorage or sessionStorage.

Client-side action: The token is used to authenticate future requests.

Security note: Avoid storing sensitive tokens in localStorage if possible, as it can be accessed via JavaScript, making it vulnerable to XSS attacks.