Mini-Project 42: Web Application Vulnerability Remediation

1. Cross-site Scripting (XSS): XSS is now considered a part of OWASP A03:2021 -Injection vulnerability. XSS attacks are a type of injection attack in which malicious scripts are injected into otherwise and trusted websites (OWASP Foundation, n.d.). According to OWASP, an attacker can use XSS to send a malicious script to an unsuspecting user. The article states that the end user's browser has no way of knowing that the script should not be trusted and will execute the script. Additionally, because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site (OWASP Foundation, n.d.). These are three risk examples of XSS. To mitigate the risk of XSS, OWASP recommends the following combination options:
    - Framework Security: Fewer XSS bugs appear in applications built with modern web frameworks. Even with a robust framework, developers need to be aware of problems that can occur, like escape hatches that frameworks use to manipulate the DOM directly, React's dangerouslySetInnerHTML without sanitizing the HTML, Angular's bypassSecurityTrustAs* functions, and many more (OWASP Foundation, n.d.)
    - XSS Defense Philosophy: This philosophy states that each variable in a web application needs to be protected; ensure that all variables go through validation and are then escaped or sanitized.
    - Output Encoding: Output Encoding is recommended when you need to display data exactly as a user typed it in safely. Variables should not be interpreted as code instead of text. Output encoding examples include "HTML Contexts", "HTML Attribute Contexts", "JavaScript Contexts", "Quoted Data Values", "CSS Contexts", and "URL Contexts". Encoding the output in these contexts reduces XSS.
    - Dangerous Contexts: There are certain areas in code where output encoding may not work. These areas are called dangerous contexts. These areas include callback functions, code handling URLs, all JavaScript event handlers, and unsafe JS functions like setTimeout().
    - HTML Sanitization: Sometimes users need to author HTML like changing the style or structure of content. Output coding, though effective in preventing XSS here, will break the intended functionality of the application. HTML Sanitization will strip dangerous HTML from a variable and return a safe string of HTML. OWASP recommends DOMPurify for HTML Sanitization.

2. Cross-site request forgery (CSRF): OWASP labels CSRF as CWE-352 and is currently part of A01:2021 -Broken Access Control vulnerability. As the label shows, A01, a broken access control vulnerability, is the number 1 vulnerability on the list. Specifically, CSRF is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated (S, n.d.). The victim is usually tricked

into submitting a malicious request that inherits the identity and privileges of the victim to perform an undesired function on the victim's behalf. The attacker uses privileges to cause a state change on the server, such as changing the victim's email address or password or purchasing something. An attacker can use CSRF to obtain the victim's private data via a special form of the attack, known as login CSRF. The attacker forces a non-authenticated user to log in to an account the attacker controls. If the victim does not realize this, they may add personal data—such as credit card information—to the account. The attacker can then log back into the account to view this data, along with the victim's activity history on the web application. In short, OWASP recommends the following to prevent CSRF:

- Check if a framework has built-in CSRF protection and use it
- For stateful software, use the synchronizer token pattern
- For stateless software, use double-submit cookies
- For API-driven sites that don't use <form> tags, consider using custom request headers
- Remember that any Cross-Site Scripting (XSS) can be used to defeat all CSRF mitigation techniques!
- Do not use GET requests for state-changing operations.

3. SQL injection: As part of A03 Injection, a CWE-89: SQL injection attack consists of the insertion or "injection" of a SQL query via the input data from the client to the application (OWASP Foundation, n.d.). A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. To avoid SQL injection flaws, developers need to either: a) stop writing dynamic queries with string concatenation and/or b) prevent user-supplied input which contains malicious SQL from affecting the logic of the executed query.
   Primary Defenses:
   - Use of Prepared Statements (with Parameterized Queries)
   - Use of Properly Constructed Stored Procedures
   - Allow-list Input Validation
   - Escaping All User-Supplied Input
   - Enforcing Least Privilege
   - Performing Allow-list Input Validation as a Secondary Defense

4. Sensitive data exposure: This is currently tracked at A02 cryptographic failures by OWASP. The focus is on failures related to the lack of cryptography which often leads to the exposure of sensitive data. This exposure can apply to data at rest or in transit. Security flaws that commonly lead to cryptography failures include:
   - Transmitting secret data in plain text
   - Use of old/less secure algorithm
   - Use of a hard-coded password in config files
   - Improper cryptographic key management

- Insufficient randomness for cryptographic functions
- Missing encryption
- Insecure implementation of certificate validation
- Use of deprecated hash functions
- Use of outdated padding methods
- The presence of sensitive data in source control
- Use of insecure initialization vectors
- Use of passwords as crypto keys without a password-based key derivation function
- Exploitable side-channel information or cryptographic error messages

These flaws can lead to major data breaches leading to significant costs to companies. To mitigate exposing sensitive data, OWASP recommends

- Catalog All Data Processed By the Application
- Discard Unused Data
- Disable Caching for Responses with Sensitive Data
- Use Appropriate Initialization Vectors
- Use Updated and Established Cryptographic Functions, Algorithms, and Protocols
- Enforce Key Rotation
- Use Authenticated Encryption Instead of Plain Encryption

5. Insufficient logging and monitoring: A09: Security Logging and monitoring is categorized to help detect, escalate, and respond to active breaches. Insufficient logging and monitoring will result in breaches going undetected. Examples of insufficient logging and monitoring are:
    - Auditable events, such as logins, failed logins, and high-value transactions, are not logged.
    - Warnings and errors generate no, inadequate, or unclear log messages.
    - Logs of applications and APIs are not monitored for suspicious activity.
    - Logs are only stored locally.
    - Appropriate alerting thresholds and response escalation processes are not in place or effective.
    - Penetration testing and scans by dynamic application security testing (DAST) tools (such as OWASP ZAP) do not trigger alerts.
    - The application cannot detect, escalate, or alert for active attacks in real-time or near real-time.

    To ensure that events are sufficiently and effectively logged and monitored, developers should implement these controls:
    - Ensure all login, access control, and server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts and held for enough time to allow delayed forensic analysis.
    - Ensure that logs are generated in a format that log-management solutions can easily process.

- Ensure log data is encoded correctly to prevent injections or attacks on the logging or monitoring systems.
- Ensure high-value transactions have an audit trail with integrity controls to prevent tampering or deletion, such as append-only database tables or similar.
- DevSecOps teams should establish effective monitoring and alerting such that suspicious activities are detected and responded to quickly.
- Establish or adopt an incident response and recovery plan, such as the National Institute of Standards and Technology (NIST) 800-61r2 or later.