DUMPSBOSS.

Certified Application Security Engineer (CASE) JAVA

ECCouncil 312-96

Version Demo

Total Demo Questions: 6

Total Premium Questions: 68

Buy Premium PDF

https://dumpsboss.co

support@dumpsboss.co

support@dumpsboss.co
dumpsboss.co

Topic Break Down

Topic	No. of Questions
Topic 1, Application Security Fundamentals	6
Topic 2, Application Security Requirements	3
Topic 3, Secure Application Design and Architecture	10
Topic 4, Secure Coding Practices	36
Topic 5, Application Security Testing	4
Topic 6, Application Security Deployment and Maintenance	9
Total	68

QUESTION NO: 1

A login page returns `Unknown username` when an account does not exist and `Incorrect password` when the password is invalid. An attacker uses these responses to identify registered accounts. Which modification is most appropriate?

A. Disable audit logging for failed authentication attempts.

B. Return a generic authentication-failure response for both unknown users and invalid passwords.

C. Return the account-lockout status and remaining attempts in every failure response.

D. Require a password reset whenever a user enters an invalid password.

ANSWER: B

Explanation:

The application should return the same generic failure response for both conditions, such as `Invalid username or password`, and should avoid meaningful timing differences where practical. This reduces the information available to an attacker attempting to enumerate valid accounts.

Disabling all logging would remove useful security evidence and does not prevent response-based enumeration. Returning a detailed account-lockout explanation can disclose even more account state. Requiring a password reset after every failed login is disruptive and is not an appropriate account-enumeration control.

QUESTION NO: 2

Identify the type of attack depicted in the following figure.

A. SQL Injection Attacks

B. Session Fixation Attack

C. Parameter Tampering Attack

D. Denial-of-Service Attack

ANSWER: C

Explanation:

Parameter Tampering Attack is correct because the figure shows request URL query-string values being deliberately changed, including values such as `debit` and `status`. Parameter tampering occurs when an attacker modifies client-controlled data sent to an application, whether in URL query parameters, form fields, cookies, hidden fields, or request bodies. The attacker's goal is to make the server process a value that was not intended by the application workflow, such as a different account identifier, transaction amount, authorization state, or business-process status.

The security issue arises when the server trusts these client-supplied values rather than deriving sensitive values from the authenticated session and enforcing authorization and business rules on the server. For example, a server must validate that the authenticated user is entitled to act on the referenced resource and that a requested state transition is permitted. Effective defenses include allow-list validation, server-side authorization checks for every request, integrity protection where appropriate, and avoiding reliance on hidden or URL-based fields for security decisions. OWASP describes the related risk of manipulating object identifiers and requests in its API Security Top 10 and recommends robust server-side access control in the Authorization Cheat Sheet.

QUESTION NO: 3

A Java web application redirects users after login using a `returnUrl` request parameter. The application must support navigation only to pages within its own site. Which validation approach best prevents an open redirect vulnerability?

A. Redirect only to a server-side allowlist of local routes and reject absolute and scheme-relative URLs.

B. Allow any URL that contains the application's hostname as a substring.

C. URL-encode the supplied value and redirect to the encoded destination.

D. Place the return URL in a hidden form field instead of a query string.

ANSWER: A

Explanation:

The application should accept only an allowlisted set of local paths or route identifiers, resolve them safely, and reject absolute URLs and scheme-relative values such as `//attacker.example`. A server-side mapping from a small set of workflow names to known local destinations is preferable to redirecting directly to arbitrary client-supplied URLs.

Checking only whether the value contains the application's hostname can be bypassed through crafted URLs. URL encoding does not establish that a destination is trusted. Hiding the parameter in a form does not prevent a client from changing it.

QUESTION NO: 4

Ted is an application security engineer who ensures application security activities are being followed during the entire lifecycle of the project. One day, he was analyzing various interactions of users depicted in the use cases of the project under inception. Based on the use case in hand, he started depicting the scenarios where attacker could misuse the application. Can you identify the activity on which Ted is working?

A. Ted was depicting abuse cases

B. Ted was depicting abstract use cases

C. Ted was depicting lower-level use cases

D. Ted was depicting security use cases

ANSWER: A

Explanation:

Ted was depicting abuse cases is correct. Abuse cases model the application from an adversarial perspective. They build on ordinary use cases—which describe legitimate actors, expected interactions, and business goals—by documenting how a malicious actor could deliberately misuse a feature, violate an assumption, bypass a control, access data without authorization, or otherwise cause harm. This is especially useful during project inception because the team can identify security-relevant behavior before detailed design and implementation decisions become expensive to change.

In this scenario, Ted first examines the users' interactions represented by the existing use cases and then derives scenarios in which an attacker misuses the application. That process is the defining purpose of abuse-case modeling. The resulting abuse cases help the project team translate anticipated misuse into security requirements, mitigations, and test scenarios. For example, an abuse case may lead to requirements for authorization checks, input validation, rate limiting, audit logging, or secure session handling. OWASP identifies threat modeling as an early, structured activity for finding threats and defining mitigations, and its secure-design guidance emphasizes addressing security requirements throughout the development lifecycle. See the OWASP Threat Modeling guide and the OWASP Proactive Controls.

QUESTION NO: 5

A payment service records failed payment references in an audit log. A tester submits a reference containing carriage-return and line-feed characters and causes a forged line to appear as a separate audit event. Which control should the developer apply before recording the reference?

A. Apply HTML encoding to the reference before writing it to the audit log.

B. Validate the reference format and reject or encode carriage-return and line-feed characters before logging it.

C. Encrypt the audit log file after each payment reference is recorded.

D. Write the complete exception stack trace whenever the reference is invalid.

ANSWER: B

Explanation:

The developer should validate the reference against its expected format and reject or safely encode control characters, especially carriage return and line feed, before logging it. This prevents untrusted data from creating false log entries or altering the structure of line-oriented logs. Structured logging is also useful because it records values as distinct fields rather than requiring string concatenation.

HTML encoding is intended for browser output and does not protect a server log. Encrypting the log file protects it at rest but does not prevent malformed entries from being written. Logging the full exception stack trace can reveal implementation details and does not address log-forging input.

QUESTION NO: 6

A Spring MVC application binds an account-update form directly to an `Account` entity. The entity includes `email`, `displayName`, and `role` fields. A normal user adds `role=ADMIN` to the request and the value is persisted. Which design change best mitigates this vulnerability?

A. Bind the request to an update DTO containing only permitted fields and explicitly map those fields to the entity.

B. Remove the role field from the HTML form while continuing to bind directly to the entity.

C. Add client-side validation that prevents the role field from being changed.

D. Make the `Account` entity implement `Serializable`.

ANSWER: A

Explanation:

The appropriate control is to bind the request to a dedicated data-transfer object containing only fields that the user is permitted to modify, then explicitly copy validated values to the persistent entity. This prevents automatic binding of sensitive properties such as `role`, account status, ownership, or privilege flags.

Hiding the role field in the HTML form does not stop a user from creating the parameter manually. Client-side validation can be bypassed. Making the entity serializable does not restrict Spring data binding. Authorization should still be enforced for the update operation, but an allowlisted request model specifically addresses over-posting or mass-assignment of entity properties.