Best App Security Practices A01-A05:

A01: Broken Access Control

1. Access Control Assessment:
    - Review access controls to identify any areas lacking proper enforcement.
    - Check for direct object references, insecure direct object references, missing function-level access controls, etc.

2. Access Control Testing:
    - Conduct comprehensive testing to verify that access controls are functioning as intended.
    - Use both automated tools and manual testing to identify potential vulnerabilities.

3. Role-Based Access Control (RBAC):
    - Implement RBAC principles to ensure users have appropriate access based on their roles and responsibilities.
    - Regularly review and update role assignments to align with business needs.

4. Session Management:
    - Implement secure session management practices to prevent session hijacking or fixation.
    - Use secure session tokens, enforce session timeouts, and implement strong session ID generation.

5. Access Logs Monitoring:
    - Monitor access logs for suspicious activities or unauthorized access attempts.
    - Implement logging mechanisms to track user actions and access control decisions.

Example Attack Scenarios:

Scenario #1: The application uses unverified data in a SQL call that is accessing account information:

```
pstmt.setString(1, request.getParameter("acct"));
ResultSet results = pstmt.executeQuery( );
```

An attacker simply modifies the browser's 'acct' parameter to send whatever account number they want. If not correctly verified, the attacker can access any user's account.

https://example.com/app/accountInfo?acct=notmyacct

Scenario #2: An attacker simply forces browses to target URLs. Admin rights are required for access to the admin page.

https://example.com/app/getappInfo
https://example.com/app/admin_getappInfo

If an unauthenticated user can access either page, it's a flaw. If a non-admin can access the admin page, this is a flaw.

A02: Cryptographic Failures

1. Cryptographic Algorithm Selection:
    - Choose cryptographic algorithms and protocols based on industry best practices and standards.
    - Avoid using weak or deprecated algorithms susceptible to attacks.

2. Secure Key Management:
    - Implement secure key management practices, including key generation, storage, rotation, and disposal.
    - Protect cryptographic keys from unauthorized access or disclosure.

3. Data Encryption:
    - Encrypt sensitive data at rest and in transit using strong encryption algorithms and protocols.
    - Implement secure communication channels such as TLS/SSL to protect data in transit.

4. Randomness and Entropy:
    - Ensure the use of high-quality random number generators for generating cryptographic keys and nonces.
    - Monitor and test entropy sources to maintain an adequate level of randomness.

5. Cryptographic Protocol Validation:
    - Validate cryptographic protocols and configurations to prevent known weaknesses or vulnerabilities.
    - Regularly review and update cryptographic implementations to address new threats and vulnerabilities.

Example Attack Scenarios:

Scenario #1: An application encrypts credit card numbers in a database using automatic database encryption. However, this data is automatically decrypted when retrieved, allowing a SQL injection flaw to retrieve credit card numbers in clear text.

Scenario #2: A site doesn't use or enforce TLS for all pages or supports weak encryption. An attacker monitors network traffic (e.g., at an insecure wireless network), downgrades connections from HTTPS to HTTP, intercepts requests, and steals the user's session cookie. The attacker then replays this cookie and hijacks the user's (authenticated) session, accessing or modifying the user's private data. Instead of the above they could alter all transported data, e.g., the recipient of a money transfer.

Scenario #3: The password database uses unsalted or simple hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password database. All the unsalted hashes can be exposed with a rainbow table of pre-calculated hashes. Hashes generated by simple or fast hash functions may be cracked by GPUs, even if they were salted.

A03: Injection

1. Input Validation:
    - Implement strict input validation to sanitize user inputs and prevent injection attacks.
    - Use input validation libraries or frameworks to handle different data types and formats securely.

2. Parameterized Queries:
    - Use parameterized queries or prepared statements to interact with databases and prevent SQL injection attacks.
    - Avoid concatenating user inputs directly into SQL queries.

3. Output Encoding:
    - Encode output data to prevent cross-site scripting (XSS) and other injection attacks.
    - Use output encoding libraries or functions appropriate for the output context (HTML, URL, JavaScript, etc.).

4. Whitelisting:
    - Implement whitelisting techniques to validate input against a list of allowed characters or patterns.
    - Reject input that does not match the expected format or criteria.

5. Static and Dynamic Analysis:
    - Perform static code analysis and dynamic testing to identify and remediate injection vulnerabilities.
    - Use automated scanning tools and manual code reviews to detect injection flaws in the application code

Example Attack Scenarios:

Scenario #1: An application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT \* FROM accounts WHERE custID='" +
request.getParameter("id") + "'";
```

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g., Hibernate Query Language (HQL)):

```
Query HQLQuery = session.createQuery("FROM accounts WHERE custID='" +
request.getParameter("id") + "'");
```

In both cases, the attacker modifies the ‘id’ parameter value in their browser to send: ' UNION SLEEP(10);--. For example:

http://example.com/app/accountView?id=' UNION SELECT SLEEP(10);--

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify or delete data or even invoke stored procedures.

.

A04: Insecure Design

1. Threat Modeling:
    - Conduct threat modeling exercises to identify potential security threats and vulnerabilities early in the design phase.
    - Consider security requirements and controls throughout the application design process.

2. Secure Architecture Patterns:
    - Follow secure architecture patterns and design principles such as defense-in-depth, least privilege, and separation of concerns.
    - Implement security controls at each layer of the application architecture.

3. Security by Design:
    - Integrate security considerations into the design process from the beginning.
    - Ensure that security features and controls are included in the initial design specifications.

4. Security Code Review:
    - Perform regular security code reviews to identify insecure design patterns or architectural flaws.
    - Use automated tools and manual inspection to analyze the application's design for potential vulnerabilities.

5. Threat Intelligence Integration:
    - Integrate threat intelligence sources into the design process to stay informed about emerging threats and attack techniques.
    - Use threat intelligence to inform design decisions and prioritize security controls.

Example Attack Scenarios:

Scenario #1: A credential recovery workflow might include “questions and answers,” which is prohibited by NIST 800-63b, the OWASP ASVS, and the OWASP Top 10. Questions and answers cannot be trusted as evidence of identity as more than one person can know the answers, which is why they are prohibited. Such code should be removed and replaced with a more secure design.

Scenario #2: A cinema chain allows group booking discounts and has a maximum of fifteen attendees before requiring a deposit. Attackers could threat model this flow and test if they could book six hundred seats and all cinemas at once in a few requests, causing a massive loss of income.

Scenario #3: A retail chain’s e-commerce website does not have protection against bots run by scalpers buying high-end video cards to resell auction websites. This creates terrible publicity for the video card makers and retail chain owners and enduring bad blood with enthusiasts who cannot obtain these cards at any price. Careful anti-bot design and domain logic rules, such as purchases made within a few seconds of availability, might identify inauthentic purchases and rejected such transactions.

A05: Security Misconfiguration

1. Configuration Management:
    - Establish robust configuration management processes to manage application configurations securely.
    - Implement version control and change management practices to track and monitor configuration changes.

2. Secure Defaults:
    - Configure applications, frameworks, and platforms with secure default settings.
    - Review and adjust default configurations to minimize the attack surface and reduce exposure to common security risks.

3. Security Patch Management:
    - Develop and implement a security patch management program to apply patches and updates promptly.

- Monitor security advisories and vendor announcements for patches addressing known vulnerabilities.

4. Continuous Configuration Auditing:
    - Use automated tools and scripts to perform continuous configuration audits and vulnerability assessments.
    - Scan for misconfigurations, insecure settings, and deviations from security best practices.

5. Least Privilege Principle:
    - Apply the principle of least privilege to restrict access permissions and privileges to the minimum necessary.
    - Enforce strict access controls and authorization mechanisms to limit access to sensitive resources.

Example Attack Scenarios:

Scenario #1: The application server comes with sample applications not removed from the production server. These sample applications have known security flaws attackers use to compromise the server. Suppose one of these applications is the admin console, and default accounts weren't changed. In that case, the attacker logs in with default passwords and takes over.

Scenario #2: Directory listing is not disabled on the server. An attacker discovers they can simply list directories. The attacker finds and downloads the compiled Java classes, which they decompile and reverse engineer to view the code. The attacker then finds a severe access control flaw in the application.

Scenario #3: The application server's configuration allows detailed error messages, e.g., stack traces, to be returned to users. This potentially exposes sensitive information or underlying flaws such as component versions that are known to be vulnerable.

Scenario #4: A cloud service provider (CSP) has default sharing permissions open to the Internet by other CSP users. This allows sensitive data stored within cloud storage to be accessed.