Secure Coding Practices Checklist

Input Validation

1. Centralized Data Validation

- **Conduct all data validation on a trusted system**: Ensure that all data validation is performed on the server-side to maintain control and security. Client-Side validation May be implemented but only for usability reasons in addition to server-side validation or to prevent client-side attacks.
- **Centralized Input Validation**: Implement a centralized input validation routine for the entire application to ensure consistency and efficiency in handling input validation.

2. Data Source Classification and Handling

- **Identify and Classify Data Sources**: Clearly identify all data sources and classify them as trusted or untrusted. Examples of untrusted sources include databases and file streams.
- **Validate Untrusted Data**: All data originating from untrusted sources must be validated rigorously before processing (including hidden form fields and HTTP header such as cookies).

3. Character Set and Encoding

- **Specify Character Sets**: Use appropriate character sets like UTF-8 for all input sources to ensure consistency.
- **Canonicalize Before Validating**: Encode data to a common character set before performing validation to avoid encoding-based attacks.
- **UTF-8 Decoding**: If the system supports UTF-8 extended character sets, ensure validation occurs after UTF-8 decoding is complete.

4. Input Validation Rules

- **Reject on Validation Failure**: Any input that fails validation should be rejected to prevent security vulnerabilities.
- **Client-Side Data Validation**: Validate all data provided by clients, including parameters, URLs, and HTTP headers, especially data from automated post-backs.
- **ASCII Header Validation**: Ensure that header values in both requests and responses contain only ASCII characters.

5. Redirect Validation

- **Validate Redirect Data**: Always validate data involved in redirects to prevent attackers from submitting malicious content directly to the redirect target.

6. Data Type and Format Validation

- **Expected Data Types**: Validate that the input data conforms to the expected data types.
- **Data Range and Length**: Validate that the input data falls within the expected range and length constraints Example:
- Numeric instead of a String datatype,
- limitations for numeric data types or
- restricted allowed characters for a string (e.g. only “a-z” and “A-Z”).
- **Whitelist Allowed Characters**: Where possible, validate input against a whitelist (positive validation model) of allowed characters to prevent injection attacks.

7. Handling Hazardous Characters

- **Additional Controls for Hazardous Characters**: If hazardous characters must be allowed, implement additional controls like output encoding and secure APIs to handle these characters safely. Common hazardous characters include: < > " ' % ( ) & + \ \' \".

8. Special Input Checks

- **Null Bytes**: Check for null bytes (%00) in the input.
- **New Line Characters**: Check for new line characters (%0d, %0a, \r, \n).
- **Path Alterations**: Check for “dot-dot-slash" sequences (../ or ..\) and address alternate representations in UTF-8 encoding such as %c0%ae%c0%ae/ using canonicalization techniques.

Output Encoding

1.Conduct Encoding on Trusted Systems

- Perform all encoding on a trusted system, such as the server.

2. Utilize Standard, Tested Routines

- Use standardized and tested routines for each type of outbound encoding to ensure consistency and reliability.

3. Contextual Output Encoding

- Contextually encode all data returned to the client that originated outside the application's trust boundary. For example, HTML entity encoding should be used where appropriate, but it may not be suitable for all contexts.

4. Encode All Characters by Default

- Encode all characters unless they are specifically known to be safe for the intended interpreter.

5. Sanitize Output for Different Query Types

- Contextually sanitize all output of untrusted data in queries for SQL, XML, and LDAP to prevent injection attacks.

6. Sanitize Output for Operating System Commands

- Ensure that all output of untrusted data to operating system commands is sanitized to avoid command injection vulnerabilities.

Authentication and Password Management

User-Authentication

1. Assurance-Based User Registration Requirements

- Implement user registration with an identification method appropriate for the assurance class of the application:

  a) **Standard Assurance:** Use e-mail addresses for identification.

  b) **High Assurance:** Use an additional factor (e.g., mobile phone) for identification.

  c) **Very High Assurance:** Require personal identification for registration.

2. Require Authentication for All Resources

- Ensure that all pages and resources are protected by authentication controls, except those specifically intended to be public.

3. Trusted System Enforcement

- Implement all authentication controls on a trusted system, such as the server, to prevent client-side manipulation.

4. Standard Authentication Services

- Utilize standard, tested authentication services whenever possible to benefit from proven security measures.

5. Centralized Authentication Implementation

- Implement a centralized system for all authentication controls, including libraries that interact with external authentication services, to maintain consistency and security.

6. Segregate Authentication Logic

- Separate authentication logic from the resource being requested. Use redirection to and from the centralized authentication control to ensure secure handling of credentials.

7. Fail Securely

- Ensure that all authentication controls are designed to fail securely, preventing any unintended access in case of failure.

8. Secure Administrative and Account Management Functions

- Ensure that administrative and account management functions are at least as secure as the primary authentication mechanism.

9. Secure Credential Storage

- If your application manages a credential store, store only cryptographically strong, one-way salted hashes of passwords. Ensure the table/file storing passwords and keys is writable only by the application. Avoid using the MD5 algorithm.

10. Password Hashing on Trusted Systems

- Implement password hashing exclusively on trusted systems, such as the server, to prevent client-side vulnerabilities.

11. Validate Authentication Data

- Validate authentication data only after all data input is complete, especially in sequential authentication implementations, to prevent partial validation attacks.

12. Generic Failure Responses

- Design authentication failure responses to avoid revealing which part of the authentication data was incorrect. Use a generic message like "Invalid username and/or password" and ensure error responses are identical in both display and source code.

13. Authenticate External System Connections

- Utilize authentication for connections to external systems that handle sensitive information or functions to prevent unauthorized access.

14. Secure Storage of External Authentication Credentials

- Encrypt and store authentication credentials for accessing external services in a protected location on a trusted system. The source code should never be used as a storage location for these credentials.

15. Use HTTP POST for Credentials Transmission

- Transmit authentication credentials only using HTTP POST requests to ensure they are not exposed in URLs or logs.

Password Management

1.User Password Compliance

- Align password length, complexity, and rotation policies with the National Institute of Standards and Technology (NIST) 800-63b guidelines in section 5.1.1 for Memorized Secrets or other modern, evidence-based password policies.

2. General Password Requirements

- Unless specified otherwise by the password policy, user passwords should meet the following requirements:
  - Length of at least 8 characters.
  - Include a mix of characters, digits, and special characters.
  - Not be identical to the username.
  - Masked on all HTML password fields.
  - Not logged or cached.
  - Encrypted when transferred over insecure channels.
  - Not transmitted in URLs.
  - Stored as a salted secure hash, ideally with key stretching using bcrypt, scrypt, PBKDF2, or Argon2 algorithms.

3. Initial Password Change

- Initial user passwords must be changed by the user at the first login.

4. Avoid Standard Passwords

- Standard passwords set by the vendor must not be used and should be replaced by strong individual passwords.

5. Secure Transmission of Passwords

- Only send non-temporary passwords over an encrypted connection or as encrypted data, such as in an encrypted email. Temporary passwords associated with email resets may be an exception.

6. Obscure Password Entry

- Password entry should be obscured on the user's screen (e.g., in web forms, use the input type "password").

7. Account Disabling on Invalid Login Attempts

- Disable accounts after a set number of invalid login attempts (e.g., five attempts), with a lockout period sufficient to discourage brute force attacks but not long enough to enable a denial-of-service attack.

8. Password Reset and Changing Controls

- Apply the same level of control for password reset and changing operations as for account creation and authentication.

9. Secure Password Reset Questions

- Use password reset questions that support sufficiently random answers (e.g., "favorite book" is a bad question due to common answers like "The Bible").

10. Email-based Resets

- For email-based resets, send emails only to a pre-registered address with a temporary link or password.

11. Temporary Passwords and Links

- Ensure temporary passwords and links have a short expiration time and enforce changing temporary passwords on the next use.

12. User Notifications

- Notify users when a password reset occurs.

13. Prevent Password Reuse

- Implement measures to prevent password reuse and ensure passwords are at least one day old before they can be changed to prevent attacks on password reuse.

14. Enforce Password Changes

- Enforce password changes as required by policy or regulation, with more frequent changes for critical systems, administratively controlled.

15. Disable "Remember Me" Functionality

- Disable "remember me" functionality for password fields.

16. Report Last Account Use

- Report the last use (successful or unsuccessful) of a user account at their next successful login.

17. Monitor for Account Attacks

- Implement monitoring to identify attacks against multiple user accounts using the same password to bypass standard lockouts when user IDs can be harvested or guessed.

18. Re-authenticate for Critical Operations

- Re-authenticate users before performing critical operations.

19. Multi-Factor Authentication (MFA)

- Use Multi-Factor Authentication for highly sensitive or high-value transactional accounts

20. Weak Password Checks:

- Implement checks for weak passwords by testing new or changed passwords against a list of the top 10,000 worst passwords.

Session Management

1.Use Built-in Session Management Controls

- Utilize the server or framework’s session management controls. Only recognize these session identifiers as valid.

2. Trusted Session Identifier Creation

- Ensure session identifier creation is always done on a trusted system, such as the server.

3. Random Session Identifiers

- Use well-vetted algorithms to generate sufficiently random session with high entropy identifiers.

4. Cookie Domain and Path Restrictions

- Set the domain and path for cookies containing authenticated session identifiers to an appropriately restricted value for the site.

5. Logout Functionality

- Fully terminate the associated session or connection upon logout.
- Ensure logout functionality is available from all pages protected by authorization.

6. Session Inactivity Timeout (TBD)

- Establish a session inactivity timeout that balances risk and business functional requirements, ideally no more than several hours.

7. Disallow Persistent Logins

- Enforce periodic session terminations, even for active sessions, especially for applications connecting to critical systems. Provide users with sufficient notification to mitigate negative impacts.

8. Session Management on Login

- Close any pre-login session and establish a new session after a successful login.
- Generate a new session identifier upon any re-authentication.

9. Prevent Concurrent Logins

- Disallow concurrent logins with the same user ID.

10. Session Identifier Security

- Avoid exposing session identifiers in URLs, error messages, or logs. Session identifiers should only be located in the HTTP cookie header. For example, do not pass session identifiers as GET parameters.

11. Server-side Session Data Protection

- Protect server-side session data from unauthorized access by other server users through appropriate access controls.

12. Periodic Session Identifier Renewal

- Periodically generate a new session identifier and deactivate the old one to mitigate session hijacking scenarios.

13. Secure Session Transition

- Generate a new session identifier if the connection security changes from HTTP to HTTPS. Consistently utilize HTTPS within an application rather than switching between HTTP and HTTPS.

14. Enhanced Session Management for Sensitive Operations

- Use per-session strong random tokens or parameters for sensitive server-side operations, such as account management, to prevent Cross-Site Request Forgery (CSRF) attacks.
- For highly sensitive or critical operations, utilize per-request, as opposed to per-session, strong random tokens or parameters.

15. Secure Cookie Attributes

- Use secure attributes for cookies, such as **HttpOnly**, **Secure**, and **SameSite**.
- Set the "secure" attribute for cookies transmitted over a TLS connection.
- Set cookies with the HttpOnly attribute unless client-side scripts within the application specifically require reading or setting a cookie's value.

Access Control

1.Use Trusted System Objects

- Use only trusted system objects, such as server-side session objects, for making access authorization decisions.

2. Centralized Access Authorization

- Use a single site-wide component to check access authorization, including libraries that call external authorization services.

3. Secure Failure of Access Controls

- Ensure that access controls fail securely.

4. Deny Access on Configuration Failure

- Deny all access if the application cannot access its security configuration information.

5. Request-Level Authorization

- Enforce authorization controls on every request, including those made by server-side scripts, "includes," and requests from rich client-side technologies like AJAX and Flash.

6. Segregation of Privileged Logic

- Segregate privileged logic from other application code.

7. Restrict Resource Access

- Restrict access to files or other resources, including those outside the application's direct control, to only authorized users.

8. Protected URLs and Functions

- Restrict access to protected URLs and functions to only authorized users.

9. Direct Object References

- Restrict direct object references to only authorized users.

10. Service Access

- Restrict access to services to only authorized users.

11. Application Data Access

- Restrict access to application data to only authorized users.

12. User and Data Attributes

- Restrict access to user and data attributes and policy information used by access controls.

13. Security Configuration Information

- Restrict access to security-relevant configuration information to only authorized users.

14. Consistency in Access Control Rules

- Ensure server-side implementation and presentation layer representations of access control rules match.

15. Client-Side State Data

- If state data must be stored on the client, use encryption and integrity checking on the server side to catch state tampering.

16. Enforce Business Rules

- Enforce application logic flows to comply with business rules.

17. Transaction Limits

- Limit the number of transactions a single user or device can perform in a given period of time. The transactions/time should exceed actual business requirements but be low enough to deter automated attacks.

18. Referer Header Checks

- Use the "referer" header as a supplemental check only; it should never be the sole authorization check, as it can be spoofed.

19. Session Re-Validation

- If long authenticated sessions are allowed, periodically re-validate a user’s authorization to ensure their privileges have not changed. If they have, log the user out and force them to re-authenticate.

20. Account Auditing

- Implement account auditing and enforce the disabling of unused accounts (e.g., after no more than 30 days from the expiration of an account’s password).

21. Account and Session Management

- Support disabling of accounts and terminating sessions when authorization ceases (e.g., changes to role, employment status, business process, etc.).

22. Least Privilege for Service Accounts

- Ensure service accounts or accounts supporting connections to or from external systems have the least privilege possible.

23. Access Control Policy

- Create an Access Control Policy to document an application's business rules, data types, and access authorization criteria and/or processes to ensure access is properly provisioned and controlled. This includes identifying access requirements for both data and system resources.

Error Handling and Logging

1.Avoid Disclosure of Sensitive Information:

- Do not include sensitive information in error responses. This includes system details, session identifiers, or account information.
- Use generic messages that do not reveal internal workings or configurations.

2. Error Handlers Configuration:

- Implement error handlers that prevent the display of debugging or stack trace information to users.
- Configure custom error pages to handle exceptions gracefully without revealing sensitive details.

3. Application Error Handling:

- Ensure the application itself handles errors, rather than relying on server configurations.
- Free allocated memory properly when errors occur to avoid memory leaks.

4. Security Control Errors:

- Deny access by default when security control-related errors occur, ensuring a secure fail state.

Logging

1.Trusted System Implementation:

- Implement all logging controls on a trusted system such as the server to ensure integrity and security of logs.

2. Logging of Security Events:

- Ensure logging controls support recording both successful and failed security events.

3. Important Log Event Data:

- Ensure logs contain crucial event data such as timestamps, user actions, and error codes.

4. Security of Log Data:

- Ensure log entries containing untrusted data cannot execute as code within the log viewing interface or software.
- Restrict log access to authorized personnel only to maintain data security.

5. Logging Practices:

- Utilize a master routine for all logging operations to maintain consistency.
- Avoid storing sensitive information such as passwords, session identifiers, or unnecessary system details in logs.

6. Log Analysis Mechanisms:

- Ensure mechanisms are in place to facilitate effective log analysis.

7. Detailed Event Logging:

- Log all input validation failures to monitor potential security threats.
- Record all authentication attempts, with a special focus on failures to detect unauthorized access attempts.
- Log all access control failures to identify potential security breaches.
- Capture all apparent tampering events, including unexpected changes to state data.
- Record attempts to connect with invalid or expired session tokens to track potential session hijacking attempts.
- Log all system exceptions to identify and resolve system issues.
- Document all administrative functions, including changes to security configuration settings to maintain an audit trail.
- Record all backend TLS connection failures to detect issues with secure communications.
- Log cryptographic module failures to identify potential weaknesses in encryption mechanisms.

8. Log Integrity:

- Use cryptographic hash functions to validate the integrity of log entries, ensuring they have not been tampered with.

Data Protection & Cryptographic Practices

Data Protection

1.Implement Least Privilege

- Restrict users to only the functionality, data, and system information required to perform their tasks.

2. Protect Cached and Temporary Data

- Protect all cached or temporary copies of sensitive data stored on the server from unauthorized access.
- Purge temporary working files as soon as they are no longer required.

3. Encrypt Highly Sensitive Information

- Encrypt highly sensitive stored information, such as authentication verification data, even on the server side.
- Use well-vetted algorithms as per "Cryptographic Practices" guidelines.

4. Protect Server-Side Source Code

- Ensure server-side source code is not downloadable by users.

5. Secure Storage of Sensitive Information

- Avoid storing passwords, connection strings, or other sensitive information in clear text or non-cryptographically secure manners on the client side.
- Refrain from embedding sensitive information in insecure formats like MS viewstate, Adobe Flash, or compiled code.

6. Remove Sensitive Comments

- Remove comments in user-accessible production code that may reveal backend systems or other sensitive information.

7. Eliminate Unnecessary Documentation

- Remove unnecessary application and system documentation that can reveal useful information to attackers.

8. Secure HTTP Requests

- Avoid including sensitive information in HTTP GET request parameters.

9. Disable Auto-Complete on Sensitive Forms

- Disable auto-complete features on forms expected to contain sensitive information, including authentication forms.

10. Control Client-Side Caching

- Disable client-side caching on pages containing sensitive information using headers like Cache-Control: no-store and Pragma: no-cache.

11. Support Data Removal

- Ensure the application supports the removal of sensitive data when it is no longer required, such as personal information or certain financial data.

12. Implement Access Controls

- Implement appropriate access controls for sensitive data stored on the server, including cached data, temporary files, and data accessible only by specific system users.

Cryptography

1.Trusted System for Cryptographic Functions

- **Implement all cryptographic functions used to protect secrets from the application user on a trusted system, such as the server.**

2. Protect Master Secrets

- Protect master secrets from unauthorized access.

3. Secure Failure of Cryptographic Modules

- Ensure cryptographic modules fail securely.

4. Use Approved Random Number Generators

- Generate all random numbers, file names, GUIDs, and strings intended to be unguessable using the cryptographic module's approved random number generator.

5. Compliance with Standards

- Ensure cryptographic modules used by the application are compliant with FIPS 140-2 or an equivalent standard.
- Refer to the validation list at NIST CMVP.

6. Key Management Policy

- Establish and utilize a policy and process for how cryptographic keys will be managed, including generation, distribution, storage, rotation, and revocation.

Database Security

1.Use Strongly Typed Parameterized Queries:

- Always use parameterized queries with strong typing to prevent SQL injection.

2. Input Validation and Output Encoding:

- Validate all input and encode all output, especially addressing meta characters. If validation fails, do not run the database command.

3. Strongly Typed Variables:

- Ensure all variables are strongly typed to avoid unexpected behavior.

4. Minimum Privilege Access:

- Access the database using the lowest level of privilege necessary.

5. Secure Credentials:

- Use secure, strong credentials for database access.

6. Secure Connection Strings:

- Do not hard code connection strings in the application. Store them in a separate, encrypted configuration file on a trusted system.

7. Use Stored Procedures:

- Abstract data access using stored procedures and remove direct permissions to base tables.

8. Close Connections Promptly:

- Close database connections as soon as they are no longer needed.

9. Strong Admin Passwords:

- Change default administrative passwords and use strong passwords or multi-factor authentication.

Database Configuration and Maintenance

10. Minimal Database Functionality:

- Disable unnecessary database functionality, services, and utility packages. Only install the features required (surface area reduction).

11. Remove Default Content:

- Remove unnecessary default vendor content, such as sample schemas.

12. Disable Unneeded Default Accounts:

- Disable any default accounts that are not required for business operations.

13. Different Credentials for Trust Levels:

- Connect to the database using different credentials for different trust levels (e.g., user, read-only user, guest, administrators).

File Management

General File Handling Security

1.Read-Only Application Files:

- Ensure application files and resources are set to read-only to prevent unauthorized modifications.

2.Virus and Malware Scanning:

- Scan all user-uploaded files for viruses and malware to protect the system from malicious content.

Handling User-Supplied Data

3. Dynamic Include Functions:

- Do not pass user-supplied data directly to any dynamic include functions to prevent remote code execution vulnerabilities.

4. Dynamic Redirects:

- Avoid passing user-supplied data into dynamic redirects. If necessary, ensure redirects only accept validated, relative path URLs.
- Use index values mapped to a pre-defined list of paths instead of directory or file paths, and never send the absolute file path to the client.

File Upload Security

1.Authentication and File Uploads:

- Require user authentication before allowing any file upload to ensure only authorized users can upload files.

2. Limiting File Types and size:

- Restrict the file-upload function by IT operations as much as possible. Specify the allowed file size, allowed file types, and allowed storage locations. Determine which clients are allowed to use the function. Set access and execution rights restrictively. Ensure that clients can only save files in the predetermined allowed storage location. Limit the types of files that can be uploaded to those necessary for business purposes to minimize risk.

3. File Type Validation:

- Validate uploaded files by checking file headers to confirm they are of the expected type. Checking the file type by extension alone is insufficient.

4. File Storage Location:

- Do not save uploaded files in the same web context as the application. Store files on a content server or in a database.

5. Execution Privileges:

- Prevent or restrict the uploading of any file that the web server might interpret. Turn off execution privileges on file upload directories.

6. Safe Uploading in UNIX:

- Implement safe uploading practices in UNIX by mounting the targeted file directory as a logical drive using the associated path or within a chrooted environment.

7. File Name and Type Validation:

- When referencing existing files, use a whitelist of allowed file names and types. Validate the parameter being passed; if it does not match an expected value, either reject it or use a hardcoded default file value.