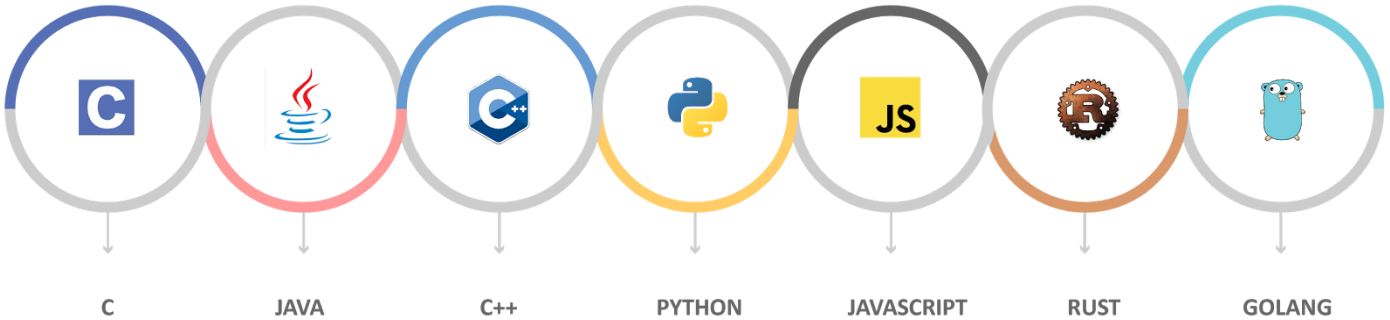
Secure Coding Practices

- OWASP Based Checklist 🌟🌟
- 200+ Test Cases 🚀🚀

Contents



1. Input Validation

The "Input validation" section in your Secure Coding Practices checklist involves measures to ensure that data coming into your application is safe and free from potential security risks. It includes practices like conducting validation on trusted systems, distinguishing between trusted and untrusted data sources, specifying character sets, validating data types, lengths, and ranges, and being cautious about potentially hazardous characters. The goal is to prevent malicious data from entering your application and to handle it safely if it must be allowed. These practices help safeguard against common security vulnerabilities, such as injection attacks, by ensuring that only well-formed, expected, and safe data is processed.

No	Test Case	Scenario	Example
1.1	Conduct all data validation on a trusted system	Ensure that all data validation is performed on the server, not on the client side. This prevents validation rules from being bypassed by manipulating client-side code.	When a user submits a form, the server checks the input data for validity and rejects it if it doesn't meet the criteria. Client-side JavaScript should not be relied upon for validation.
1.2	Identify all data sources and classify them	Classify data sources as trusted (e.g., internal databases) or untrusted (e.g., user input, external APIs). Validate all data from untrusted sources to prevent malicious input.	When processing data from user-submitted forms and external APIs, ensure that all input is thoroughly validated, regardless of its source.
1.3	Centralized input validation routine	Implement a centralized input validation routine to ensure consistent and thorough validation across the application.	Create a single validation function that is called for all user input, ensuring that every input is consistently validated according to predefined rules.

1.4	Specify proper character sets	Define a consistent character set, such as UTF-8, for all sources of input to avoid encoding mismatches and potential security vulnerabilities.	Ensure that all data, regardless of its source, is consistently encoded using UTF-8 to prevent encoding-related issues.
1.5	Encode data to a common character set	Convert data to a common character set (e.g., UTF-8) before validation to ensure consistent handling and prevent encoding-based attacks.	Before validating user input, convert it to UTF-8 to standardize character encoding.
1.6	All validation failures result in input rejection	Upon validation failure, reject the input and prevent further processing. This prevents malicious data from entering the system.	If a user submits invalid input, the system should reject it and not proceed with any further actions.
1.7	Determine system support for UTF-8 extended character sets	Determine if the system supports UTF-8 extended character sets. If supported, validate input after UTF-8 decoding is completed.	If the application supports extended character sets, ensure that validation occurs after decoding, addressing potential encoding-related vulnerabilities.
1.8	Validate all client-provided data	Validate all input from the client, including parameters, URLs, HTTP headers, and automated postbacks, to prevent malicious code injection.	When processing data from client-side sources, such as form submissions and HTTP headers, validate it rigorously to avoid code injection vulnerabilities.

1.9	Verify header values contain only ASCII characters	Ensure that header values in both requests and responses contain only ASCII characters to prevent potential attacks exploiting character encoding differences.	Check that header values are limited to ASCII characters to prevent security issues stemming from character encoding variations.
1.10	Validate data from redirects	Validate data from redirects to prevent malicious content from being directly submitted to the redirect target, bypassing application logic.	When handling data from redirects, ensure that it is validated to prevent security bypasses.
1.11	Validate for expected data types	Verify that input matches the expected data type, such as integer, string, or date, to prevent type-related errors and potential vulnerabilities.	Check that input is of the correct data type to avoid errors and security issues caused by data type mismatches.
1.12	Validate data range	Check that input values fall within the expected range to prevent overflow or underflow conditions that could lead to vulnerabilities.	Ensure that input values are within specified ranges to prevent security issues related to data overflow or underflow.
1.13	Validate data length	Limit the length of input to prevent buffer overflows and other length-based attacks.	Restrict the length of input data to mitigate security risks associated with buffer overflows and other length-based vulnerabilities.

1.14	Validate all input against a "white" list	Whenever possible, validate input against a whitelist of allowed characters to restrict the range of acceptable input and prevent malicious code injection.	Ensure that input is validated against a predefined whitelist of permissible characters to minimize security risks.
1.15	Handle hazardous characters < > " ' % ( ) & + \ ' "	If potentially hazardous characters must be allowed as input, implement additional controls like output encoding and secure APIs.	If your application requires the use of hazardous characters, apply additional security measures like output encoding and secure APIs to mitigate potential risks.
1.16	Check specific inputs	If standard validation fails, check for specific problematic characters, including null bytes (%00), new line characters (%0d, %0a, \r, \n), path alteration characters (../ or ..), and alternate representations of hazardous characters.	If standard validation doesn't catch certain inputs, inspect for problematic characters like null bytes, new line characters, path alteration sequences, or alternate encodings of hazardous characters.

2. Output Encoding

The "output encoding" section in your Secure Coding Practices checklist focuses on ensuring that data leaving your application is secure and properly formatted. It involves practices such as conducting encoding on trusted systems, utilizing established encoding routines, and contextually encoding or sanitizing data before returning it to clients. This helps prevent security vulnerabilities by ensuring that all data is presented in a safe and well-structured manner, especially when it originated from untrusted sources. Proper output encoding safeguards against issues like cross-site scripting (XSS) and injection attacks by ensuring that data is correctly processed for the intended interpreter and presentation medium.

No	Test Case	Scenario	Example
2.1	Conduct all encoding on a trusted system	Ensure that all encoding of output data is performed on the server, not on the client side, to prevent tampering and security vulnerabilities.	When generating HTML for a web page, the server should encode special characters to prevent cross-site scripting (XSS) vulnerabilities.
2.2	Utilize a standard, tested routine for each type of outbound encoding	Employ well-tested and standardized encoding routines for different data types, such as HTML entity encoding, to ensure consistent and secure encoding.	Use a trusted library or built-in functions for encoding data, such as HTML entity encoding for web content.
2.3	Contextually output encode all data returned to the client	Apply output encoding to all data that originates outside the application's trust boundary and is returned to the client. Use appropriate encoding based on the context.	When displaying user-generated content on a web page, use HTML entity encoding to prevent HTML injection.

2.4	Encode all characters unless they are known to be safe	Encode all characters in output data unless they are explicitly known to be safe for the intended interpreter to prevent malicious code injection or data manipulation.	Encode special characters like "<" and ">" in user-generated content to prevent XSS attacks.
2.5	Contextually sanitize output of untrusted data to queries	Sanitize all output of untrusted data to prevent injection attacks in queries for SQL, XML, and LDAP. Apply contextual sanitation based on the query type.	When constructing an SQL query from user input, sanitize the input to prevent SQL injection, ensuring that the user input doesn't contain harmful SQL commands.
2.6	Sanitize output of untrusted data to operating system commands	Sanitize all output of untrusted data before incorporating it into operating system commands to prevent malicious code execution or system compromise.	When executing shell commands with user input, sanitize the input to ensure that it doesn't contain harmful commands that could compromise the system.

3. Authentication and Password Management

"Authentication and Password Management" focuses on ensuring secure access to your application. It includes practices like requiring authentication for most resources, using standard authentication services, securely storing and handling passwords, enforcing password complexity and reset policies, and monitoring for suspicious activities. The goal is to protect user accounts, data, and sensitive functions by implementing strong authentication and password management practices, ultimately safeguarding against unauthorized access and security breaches.

No	Test Case	Scenario	Example
3.1	Require authentication for all pages and resources	Ensure that authentication is required for all pages and resources, except those specifically intended to be public.	All users accessing the application should be required to authenticate themselves, except for publicly accessible information like a homepage.
3.2	Enforce all authentication controls on a trusted system	Implement all authentication controls on the server, not on the client side, to prevent tampering with authentication logic.	Authentication checks should be performed on the server to ensure the security of the process.
3.3	Establish and use standard, tested authentication services	Utilize established and tested authentication services whenever possible, such as Google Sign-In or OAuth, to simplify secure authentication.	Implementing authentication through widely recognized and tested services like Google Sign-In enhances security and user experience.

3.4	Use a centralized implementation for all authentication controls	Implement a centralized authentication mechanism for all authentication-related processes, providing a single point of control for managing user credentials and access permissions.	Authentication should be handled through a centralized system that manages user credentials and permissions.
3.5	Segregate authentication logic from requested resources	Separate authentication logic from the resource being requested and use redirection to and from the centralized authentication control to ensure security.	Authentication processes should not be mixed with resource processing, and redirection to a centralized authentication control should be used.
3.6	Ensure secure failure of authentication controls	Authentication controls should fail securely, with responses not revealing which part of the authentication data was incorrect.	If authentication fails, the system should provide a generic message like "Invalid username and/or password" without indicating which part of the authentication data was incorrect.
3.7	Secure administrative and account management functions	Apply the same level of security controls to administrative and account management functions as to the primary authentication mechanism to ensure consistent security.	Administrative actions and account management should have the same level of security as user authentication.

3.8	Secure storage of password hashes	If the application manages a credential store, ensure that only cryptographically strong one-way salted hashes of passwords are stored. The table/file storing the passwords and keys should be writable only by the application.	Passwords should be stored securely using strong hashing algorithms, and access to the storage should be tightly controlled.
3.9	Implement password hashing on a trusted system	Perform password hashing on the server, not on the client side, to protect the hashing algorithm and salt values from exposure.	Password hashing should occur on the server to maintain the security of the hashing process.
3.10	Validate authentication data after all data input	Validate the authentication data only after all data input is complete, especially for sequential authentication implementations.	In a multi-step authentication process, data should be validated only after all steps are completed.
3.11	Secure authentication failure responses	Authentication failure responses should not indicate which part of the authentication data was incorrect.	Error responses in case of authentication failure should provide the same message, such as "Invalid username and/or password," without specifying the exact error.

3.12	Use authentication for connections to external systems	Utilize authentication for connections to external systems involving sensitive information or functions.	When accessing external systems with sensitive data, ensure that authentication is required to establish secure connections.
3.13	Encrypt and store external authentication credentials	Authentication credentials for accessing services external to the application should be encrypted and stored in a protected location on a trusted system, not in the source code.	External service credentials should be securely stored and not kept in the application's source code.
3.14	Use only HTTP POST requests for authentication credentials	Transmit authentication credentials using only HTTP POST requests to prevent exposure in URL parameters, which can be easily intercepted.	User credentials should be sent using HTTP POST requests to avoid exposing them in URLs.
3.15	Send non-temporary passwords over encrypted connections	Only send non-temporary passwords over encrypted connections or as encrypted data to protect them from interception.	Non-temporary passwords should be transmitted securely to prevent eavesdropping.
3.16	Enforce password complexity requirements	Enforce password complexity requirements based on policy or regulations to enhance password security.	Passwords should meet complexity requirements, such as including alphabetic, numeric, and special characters.

3.17	Enforce password length requirements	Enforce password length requirements based on policy or regulations to ensure stronger passwords.	Passwords should meet length requirements, which may include a minimum length of eight characters or more.
3.18	Obscure password entry on the user's screen	Password entry should be obscured on the user's screen, using input type "password" on web forms.	When users enter passwords, the input should be masked to prevent visual observation.
3.19	Enforce account disabling after invalid login attempts	Implement account disabling after a set number of invalid login attempts to deter brute force attacks.	After a specified number of failed login attempts, user accounts should be temporarily disabled.
3.20	Secure password reset and change operations	Password reset and change operations should have the same level of security controls as account creation and authentication.	The processes for resetting and changing passwords should be as secure as initial account creation and authentication.
3.21	Use random security questions	Password reset questions should support sufficiently random answers to enhance security.	Instead of using easily guessable questions like "favorite book," use questions that have less predictable answers.
3.22	Send reset information to pre-registered email addresses	If using email-based resets, only send email to pre-registered addresses with a temporary link or password.	When resetting passwords via email, send reset information only to email addresses that are pre-registered by the user.

3.23	Set short expiration times for temporary passwords	Temporary passwords and links should have a short expiration time to reduce the risk of unauthorized access if they are compromised.	Temporary passwords should only be valid for a short period, such as 24 hours.
3.24	Enforce changing of temporary passwords on first use	Require users to change temporary passwords immediately upon first use to prevent prolonged use of insecure temporary passwords.	Users should be prompted to change temporary passwords the first time they log in.
3.25	Notify users of password resets	Inform users when a password reset occurs to alert them of potential unauthorized access.	Users should receive notifications when their passwords are reset to keep them informed about security events.
3.26	Prevent password reuse	Implement mechanisms to discourage users from reusing old passwords to promote stronger and unique passwords.	Users should be prevented from using their old passwords when creating new ones.
3.27	Enforce a waiting period for password changes	Implement a delay between password creation and the first password change to prevent attackers from immediately changing the password.	Users should not be allowed to change their password immediately after creating it.

3.28	Enforce periodic password changes	Implement a policy requiring periodic password changes, with the time between resets being administratively controlled.	Passwords should be changed at regular intervals as specified by the policy.
3.29	Disable "remember me" functionality for password fields	Disable the "remember me" option for password fields, especially on public or shared devices to enhance security.	Users should not be given the option to have their password remembered on public or shared devices.
3.30	Report last account activity to users	Notify users of the last successful or unsuccessful login attempt to keep them informed about account activity.	Users should be aware of the last login attempts to detect any suspicious activity.
3.31	Implement monitoring for multi-account attacks	Monitor and identify attacks against multiple user accounts using the same password, a common pattern used to bypass standard lockouts.	The system should detect and prevent attacks where multiple accounts are targeted using the same password.
3.32	Change default passwords and user IDs	Change all vendor-supplied default passwords and user IDs or disable the associated accounts to prevent unauthorized access.	Vendor-supplied default credentials should be replaced or disabled to prevent unauthorized access.

3.33	Re-authenticate users before critical operations	Require users to re-authenticate before performing critical operations, adding an extra layer of security for sensitive actions.	Users should re-enter their credentials before performing critical operations like changing account settings.
3.34	Implement Multi-Factor Authentication for sensitive accounts	Use Multi-Factor Authentication (MFA) for highly sensitive or high-value transactional accounts to add an additional layer of security.	For highly sensitive accounts, require MFA to verify the user's identity.
3.35	Inspect third-party authentication code	If using third-party code for authentication, inspect the code carefully to ensure it is not affected by any malicious code.	When using third-party authentication code, thoroughly review it to check for vulnerabilities or malicious code.

4. Session Management

"Session Management" is all about ensuring secure and well-controlled user sessions within your application. It involves practices like using trusted session management controls, setting appropriate timeout periods, disallowing concurrent logins, and protecting session data from unauthorized access. The primary goal is to safeguard user sessions from common threats like session hijacking or Cross-Site Request Forgery (CSRF) by implementing robust session management practices. These practices help maintain the integrity and security of user sessions while allowing for smooth and secure interactions with your application.

No	Test Case	Scenario	Example
4.1	Use Server or Framework Session Management	Utilize the built-in session management provided by your web server or framework. This ensures that your application recognizes and uses valid session identifiers.	If you're using a web framework like Express.js in Node.js, use its session management features.
4.2	Create Session Identifiers on a Trusted System	Generate session identifiers (like cookies) on the server to prevent tampering.	When a user logs in, the server generates a unique session ID and sends it to the user's browser.
4.3	Use Secure Session Algorithms	Choose strong, secure algorithms for generating session identifiers. They should be random and unpredictable.	Instead of using a simple incremental number, use a cryptographically secure random string as the session identifier.
4.4	Set Domain and Path for Cookies	Limit where cookies with session IDs can be used by setting their domain and path.	If your site is "example.com," restrict the cookie's domain to ".example.com" to prevent it from being used on other subdomains.

4.5	Properly Terminate Sessions on Logout	Ensure that the logout function fully terminates the user's session or connection.	When a user clicks "Logout," their session is invalidated, and they are redirected to the login page.
4.6	Accessible Logout	Provide a logout option on all pages protected by authorization, so users can easily log out.	Include a "Logout" link in the navigation menu on every page.
4.7	Session Timeout	Set a short session inactivity timeout, usually a few hours, to balance security and user convenience.	Automatically log the user out after 15 minutes of inactivity.
4.8	Avoid Persistent Logins	Do not allow users to stay logged in indefinitely. Log them out periodically, even during an active session.	Prompt the user to re-enter their password after a set time, even if they are actively using the app.
4.9	Close Old Session on Login	If a user had a session before login, close that session and create a new one after successful login.	When a user logs in, their previous session becomes invalid.
4.10	Generate New Session Identifier on Re-authenticati on	Whenever a user re-authenticates (e.g., changing their password), generate a new session identifier.	When a user updates their password, their session ID changes.
4.11	No Concurrent Logins with the Same User ID	Prevent multiple simultaneous logins with the same user ID.	If a user is already logged in and tries to log in again from a different device, the previous session is invalidated.

4.12	Hide Session Identifiers	Do not expose session identifiers in URLs, error messages, or logs. Store them securely in the HTTP cookie header.	Avoid displaying session IDs in the URL like "example.com?session=123."
4.13	Protect Server-Side Session Data	Implement access controls on the server to prevent unauthorized access to session data by other users or processes on the server.	Only authorized server-side code can access and modify session data.
4.14	Rotate Session Identifiers	Periodically generate new session identifiers and deactivate the old ones. This prevents session hijacking if the original identifier was compromised.	Every hour, generate a new session ID and invalidate the previous one.
4.15	Switch to HTTPS	Change the session identifier if the connection security switches from HTTP to HTTPS. Maintain HTTPS consistently within your application.	If a user logs in via an insecure HTTP connection, generate a new session ID when they switch to a secure HTTPS connection.
4.16	Use Strong Tokens for Sensitive Operations	For sensitive server-side actions (e.g., account management), use strong random tokens or parameters to prevent Cross-Site Request Forgery (CSRF) attacks.	When a user requests to change their password, generate a unique token to ensure the request is legitimate.

4.17	Use Strong Tokens for Critical Operations	For highly sensitive or critical operations, use per-request, rather than per-session, strong random tokens or parameters.	When processing a financial transaction, generate a unique token for each step of the transaction.
4.18	Set "Secure" Attribute for Cookies	Mark cookies as "secure" when transmitted over a TLS (HTTPS) connection to prevent interception.	Ensure that cookies carrying session information are marked as "secure" to be transmitted only over secure HTTPS connections.
4.19	Use HttpOnly for Cookies	Set cookies with the HttpOnly attribute unless you specifically need client-side scripts to read or set a cookie's value.	Use HttpOnly to prevent client-side scripts from accessing cookies containing sensitive session information.

5. Access Control

"Access Control" is about ensuring that only authorized users can access specific parts of your application. This section includes practices such as using trusted system objects for authorization decisions, enforcing access controls on every request, segregating privileged logic, and restricting access to files, resources, URLs, functions, and data. The primary goal is to prevent unauthorized access to sensitive areas of your application and data by implementing strong access control practices. These practices help protect your application from security threats, ensuring that users can only access what they are allowed to and that sensitive information remains secure.

No	Test Case	Scenario	Example
5.1	Use only trusted system objects for access authorization	Use only trusted system objects, such as server-side session objects, for making access authorization decisions.	Access authorization decisions should be based on information from trusted sources within the application.
5.2	Implement a single site-wide component for access authorization	Use a single site-wide component to check access authorization, including libraries that call external authorization services.	Access to resources should be controlled consistently through a central component.
5.3	Secure failure of access controls	Access controls should fail securely, ensuring that unauthorized users cannot gain access to restricted resources.	If a user is not authorized to access a resource, they should not be able to bypass the access control mechanism.

5.4	Deny all access if security configuration information is unavailable	Deny all access if the application cannot access its security configuration information, ensuring that security settings are enforced even when configuration data is inaccessible.	If the application is unable to access its security configuration, it should default to denying access.
5.5	Enforce authorization controls on every request	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.	Authorization checks should be consistently applied to all types of requests.
5.6	Segregate privileged logic from other application code	Segregate privileged logic from other application code to prevent unauthorized access to sensitive operations.	Critical functionality that requires elevated permissions should be isolated from regular application code.
5.7	Restrict access to files and resources to authorized users	Restrict access to files and resources, including those outside the application's control, to only authorized users.	Only users with the appropriate permissions should be able to access files and resources.

5.8	Restrict access to protected URLs to authorized users	Ensure that access to protected URLs is restricted to only authorized users.	Users should be required to authenticate before accessing protected URLs.
5.9	Restrict access to protected functions to authorized users	Restrict access to protected functions to only authorized users.	Only users with proper permissions should be able to execute protected functions.
5.10	Restrict direct object references to authorized users	Limit access to direct object references to only authorized users.	Users should not be able to access objects directly if they are not authorized.
5.11	Restrict access to services to authorized users	Ensure that access to services is restricted to only authorized users.	Services should not be accessible to unauthorized users.
5.12	Restrict access to application data to authorized users	Limit access to application data to only authorized users.	Users should have access only to the data they are authorized to see or modify.
5.13	Restrict access to user and data attributes to authorized users	Control access to user and data attributes and policy information by authorized users.	User attributes and sensitive data should be restricted to authorized users.
5.14	Restrict access to security-relevant configuration information	Limit access to security-relevant configuration information to authorized users.	Configuration settings related to security should not be accessible to unauthorized users.

5.15	Ensure consistency between server-side access control rules	Ensure that server-side implementation and presentation layer representations of access control rules match.	Access control rules should be consistent between server-side implementation and the presentation layer.
5.16	Use encryption and integrity checking for client-stored state data	If state data must be stored on the client, use encryption and integrity checking on the server side to prevent tampering.	State data stored on the client side should be encrypted and protected to prevent unauthorized modifications.
5.17	Enforce compliance with business rules in application logic flows	Enforce application logic flows to comply with business rules, ensuring that application behavior aligns with business requirements.	Application logic should adhere to business rules to maintain consistency and security.
5.18	Limit the number of transactions per user or device	Limit the number of transactions a single user or device can perform within a given period to deter automated attacks.	Implement transaction limits to prevent abuse by automated systems or malicious users.
5.19	Use the "referer" header as a supplemental check	Use the "referer" header as a supplemental check, but not as the sole authorization check, as it can be spoofed.	The "referer" header can be used as an additional security measure but should not be the only means of authorization.

5.20	Periodically re-validate user authorization	If long authenticated sessions are allowed, periodically re-validate a user's authorization to ensure their privileges have not changed. Log the user out and force re-authentication if necessary.	Users with long sessions should have their authorization periodically re-validated to ensure that their privileges have not changed.
5.21	Implement account auditing and disabling of unused accounts	Implement account auditing and enforce the disabling of unused accounts, such as after 30 days from the expiration of an account's password.	Accounts that are not actively used should be disabled or removed to maintain security.
5.22	Support disabling of accounts and session termination	The application must support disabling of accounts and terminating sessions when authorization ceases, such as changes to roles, employment status, or business processes.	Accounts and sessions should be disabled when users no longer have authorization, such as when they change roles or leave the organization.

5.23	Assign least privilege to service accounts	Service accounts or accounts used for connections to external systems should have the least privilege necessary.	Service accounts should have only the permissions required for their specific tasks.
5.24	Create an Access Control Policy to document access criteria	Create an Access Control Policy to document an application's business rules, data types, and access authorization criteria or processes to ensure proper access provisioning and control.	An Access Control Policy should outline the access requirements for data and system resources, defining who can access what and under what conditions.

6. Cryptographic Practices

The "Cryptographic Practices" section in your Secure Coding Practices checklist revolves around securing data and secrets through proper cryptographic techniques. It includes practices like implementing cryptographic functions on trusted systems, safeguarding master secrets, and ensuring cryptographic modules meet recognized standards like FIPS 140-3. The primary goal is to protect sensitive information by applying sound cryptographic principles, such as generating secure random values and effectively managing cryptographic keys. These practices help guard against data breaches and unauthorized access to confidential data, enhancing the overall security of your application.

No	Test Case	Scenario	Example
6.1	Implement cryptographic functions on a trusted system	All cryptographic functions used to protect secrets from the application user must be implemented on a trusted system, such as the server.	Cryptographic operations should be performed on a trusted server rather than on the client side to prevent potential security risks.
6.2	Protect master secrets from unauthorized access	Protect master secrets used in cryptographic operations from unauthorized access, ensuring they are not exposed to potential attackers.	Sensitive cryptographic keys and master secrets should be securely stored and protected against unauthorized access.
6.3	Ensure cryptographic modules fail securely	Cryptographic modules should fail securely to prevent potential security vulnerabilities in case of errors or attacks.	Cryptographic modules should be designed to handle errors or attacks in a way that doesn't compromise security.

6.4	Use approved random number generators for un-guessable values	All random numbers, random file names, random GUIDs, and random strings should be generated using the cryptographic module's approved random number generator when these random values are intended to be un-guessable.	To create unpredictable values, use approved cryptographic random number generators rather than relying on standard random functions.
6.5	Comply with cryptographic standards	Cryptographic modules used by the application should be compliant with standards such as FIPS 140-3 or an equivalent standard.	Cryptographic modules should meet established security standards to ensure their reliability and security.
6.6	Implement key management policies and processes	Establish and utilize a policy and process for how cryptographic keys will be managed to maintain the security and integrity of cryptographic operations.	Clearly defined policies and processes for managing cryptographic keys should be in place to ensure their proper use and security.

7. Error Handling and Logging

"Error Handling and Logging" is essential for maintaining the security and integrity of your application. This section includes practices like not revealing sensitive information in error responses, implementing custom error pages, and securely handling errors, as well as logging important security events and failures. The primary goal is to enhance application security by carefully managing how errors are handled and logged. Proper error handling helps prevent attackers from exploiting vulnerabilities and allows you to monitor and analyze security events effectively to detect and respond to potential threats.

No	Test Case	Scenario	Example
7.1	Do not disclose sensitive information in error responses	Error responses should not reveal sensitive information like system details, session identifiers, or account information.	Instead of displaying detailed error messages that could expose sensitive information, provide a generic error message like "An error occurred, please try again later."
7.2	Use error handlers that do not display debugging or stack trace information	Error handlers should be configured not to reveal debugging or stack trace information to potential attackers.	Avoid showing stack trace information in error messages, as it can provide insights into the application's internal workings.
7.3	Implement generic error messages and use custom error pages	Use custom error pages with generic error messages to prevent leaking sensitive information to users or attackers.	Instead of displaying specific error messages, show custom error pages with general messages like "Page not found" or "Server error."

7.4	Handle application errors without relying on server configuration	The application should handle its errors independently without relying solely on server configuration.	Implement custom error handling logic within the application to manage and respond to errors effectively.
7.5	Free allocated memory properly when error conditions occur	Ensure that allocated memory is correctly freed when error conditions occur to prevent memory leaks.	When an error is detected, release memory resources that were allocated to avoid memory leaks.
7.6	Security controls in error handling logic should deny access by default	Error handling logic associated with security controls should deny access by default to avoid potential security vulnerabilities.	Configure security controls to deny access by default when encountering errors or unexpected conditions.
7.7	Implement logging controls on a trusted system	All logging controls should be implemented on a trusted system, such as the server.	Ensure that log entries are generated and stored on a secure and trusted system to prevent tampering or unauthorized access.
7.8	Support logging for both success and failure of specified security events	Logging controls should record both successful and failed security events for comprehensive security monitoring.	Log successful security events, like user authentication, along with failed events for a complete audit trail.

7.9	Ensure logs contain important log event data	Log entries should contain essential event data to help in diagnosing and understanding security incidents.	Include relevant information such as timestamps, user identities, and event descriptions in log entries.
7.10	Prevent log entries with un-trusted data from executing as code	Ensure that log entries containing untrusted data cannot execute as code in log viewing interfaces or software.	Implement proper encoding or escaping to prevent log entries with user input from being interpreted as executable code.
7.11	Restrict access to logs to authorized individuals	Only authorized personnel should have access to logs, ensuring the confidentiality and integrity of log data.	Control access to log files and systems containing logs to limit viewing privileges to authorized users.
7.12	Utilize a master routine for all logging operations	Implement a centralized routine for all logging operations to maintain consistency and reliability.	Use a common function or routine for logging across the application to ensure uniformity and simplify maintenance.
7.13	Do not store sensitive information in logs	Avoid storing sensitive information like system details, session identifiers, or passwords in log entries.	Ensure that logs do not contain sensitive data that could be exploited if the logs were accessed by unauthorized parties.

7.14	Ensure mechanisms exist for log analysis	Implement mechanisms to conduct log analysis and monitoring of security events to detect anomalies or potential threats.	Use log analysis tools and processes to regularly review and monitor log entries for security events.
7.15	Log all input validation failures	Record log entries for input validation failures to track and investigate potential security issues.	Capture instances where input data fails validation checks, which may indicate security vulnerabilities.
7.16	Log all authentication attempts, especially failures	Record log entries for authentication attempts, particularly failures, to monitor potential unauthorized access attempts.	Keep a record of all authentication attempts, with a focus on failed attempts, to detect and respond to potential security threats.
7.17	Log all access control failures	Record log entries for access control failures to monitor and address unauthorized access incidents.	Document cases where access control mechanisms fail to prevent unauthorized access or actions.
7.18	Log all apparent tampering events, including unexpected changes to state data	Log events that indicate potential tampering or unexpected changes to the application's state data.	Monitor and log suspicious events, like unauthorized changes to data or configuration settings.

7.19	Log attempts to connect with invalid or expired session tokens	Record log entries for attempts to connect with invalid or expired session tokens to detect potential session-related security threats.	Capture and investigate any attempts to use session tokens that are no longer valid or have been tampered with.
7.20	Log all system exceptions	Record log entries for system exceptions to identify and troubleshoot potential issues.	Capture and investigate system exceptions that may indicate problems within the application.
7.21	Log all administrative functions, including changes to security configuration settings	Document log entries for administrative functions, especially changes to security configuration settings.	Maintain an audit trail of administrative actions, especially those related to security settings, for accountability and monitoring purposes.
7.22	Log backend TLS connection failures	Record log entries for backend TLS (Transport Layer Security) connection failures to detect potential security issues with secure communication.	Monitor and log any failures in secure connections between components of the application.
7.23	Log cryptographic module failures	Document log entries for cryptographic module failures to identify and respond to potential cryptographic issues.	Capture events indicating failures or issues related to cryptographic modules used by the application.

7.24	Use cryptographic hash functions to validate log entry integrity	Employ cryptographic hash functions to verify the integrity of log entries and detect tampering.	Use hash functions to create checksums of log entries and validate their integrity during log analysis.

8. Data Protection

"Data Protection" is crucial for securing sensitive information within your application. This section involves practices like implementing the principle of least privilege, encrypting sensitive data, protecting cached or temporary files, and safeguarding server-side source code. The primary goal is to ensure the confidentiality and integrity of data, preventing unauthorized access or data leaks. Effective data protection practices, like encryption and access controls, help keep sensitive information safe from threats and unauthorized disclosure, ultimately enhancing the security of your application.

No	Test Case	Scenario	Example
8.1	Implement least privilege, restrict users to only the required functionality, data, and system information	Restrict users to only the functionality, data, and system information necessary for them to perform their tasks.	Users in a healthcare application should only have access to patient records they are authorized to view and not the entire database.
8.2	Protect cached or temporary copies of sensitive data on the server from unauthorized access and purge them when no longer needed	Ensure that cached or temporary copies of sensitive data are protected from unauthorized access, and delete them as soon as they are no longer required.	Delete cached copies of sensitive user data after the user logs out or when the data is no longer needed for processing.
8.3	Encrypt highly sensitive stored information, like authentication verification data, even on the server side	Use strong encryption for highly sensitive stored information, such as authentication verification data, even on the server.	Store user passwords using a secure hashing algorithm like bcrypt to protect them from unauthorized access.

8.4	Protect server-side source code from being downloaded by a user	Ensure that server-side source code is protected from being downloaded by users.	Use appropriate access controls and server configurations to prevent users from accessing server-side source code.
8.5	Do not store passwords, connection strings, or other sensitive information in clear text or insecure formats on the client side	Avoid storing sensitive information like passwords or connection strings in clear text or insecure formats on the client side, such as MS ViewState, Adobe Flash, or compiled code.	Store sensitive information in a secure manner using encryption and secure storage practices.
8.6	Remove comments in user-accessible production code that may reveal sensitive information	Remove comments in user-accessible production code that could reveal backend systems or sensitive information.	Delete comments in the code that contain details about the application's architecture or system configurations.
8.7	Remove unnecessary application and system documentation that could reveal information to attackers	Remove any unnecessary documentation from the application and system that may reveal useful information to potential attackers.	Avoid publishing documentation that provides insights into the application's internal workings or configurations.

8.8	Do not include sensitive information in HTTP GET request parameters	Avoid sending sensitive information in HTTP GET request parameters as it may be exposed in URLs or logs.	Instead of sending sensitive data as parameters in a URL, use HTTP POST requests or other secure methods.
8.9	Disable auto-complete features on forms with sensitive information, including authentication	Disable auto-complete features on forms that may contain sensitive information, including login and authentication forms.	Prevent browsers from automatically filling in sensitive data, such as usernames and passwords, to enhance security.
8.10	Disable client-side caching on pages with sensitive information	Disable client-side caching on pages containing sensitive information to prevent data from being stored locally.	Set appropriate HTTP headers like "Cache-Control: no-store" to instruct browsers not to cache sensitive pages.
8.11	Support the removal of sensitive data when no longer required	Implement functionality to delete sensitive data when it is no longer needed, such as personal information or certain financial data.	Allow users to request the removal of their personal data from the application when it's no longer necessary.

8.12	Implement access controls for sensitive data stored on the server, including cached data, temporary files, and data accessible only by specific users	Ensure that sensitive data stored on the server, including cached data and temporary files, is protected by appropriate access controls.	Implement access controls to restrict access to cached data and temporary files to only specific system users or roles.

9. Communication Security

"Communication Security" is all about safeguarding data as it travels between different components of your application and external systems. This section includes practices like implementing encryption for sensitive information transmission, ensuring valid and up-to-date TLS certificates, and preventing insecure fallback from failed TLS connections. The main goal is to protect data while it's in transit, ensuring that it remains confidential and integral during communication. Strong communication security practices, like TLS encryption and correct certificate management, help prevent eavesdropping and tampering with sensitive information, ultimately enhancing your application's overall security.

No	Test Case	Scenario	Example
9.1	Implement encryption for the transmission of all sensitive information, including TLS for protecting the connection	Ensure that all sensitive information is transmitted using encryption, such as TLS (Transport Layer Security), to protect the data during transmission.	When a user logs into an online banking application, their username and password are transmitted securely using TLS to prevent eavesdropping.
9.2	Ensure TLS certificates are valid, have the correct domain name, not expired, and installed with intermediate certificates when required	Validate that TLS certificates used for securing connections are valid, have the correct domain name, are not expired, and include intermediate certificates when necessary.	An e-commerce website's TLS certificate should have the correct domain name, should not be expired, and should include all required intermediate certificates to establish a secure connection.

9.3	Prevent failed TLS connections from falling back to an insecure connection	Ensure that when a TLS connection fails, it does not fall back to an insecure or unencrypted connection, which could expose sensitive data.	If a client fails to establish a TLS connection, it should not proceed with an unencrypted connection but should display an error message instead.
9.4	Use TLS connections for all content requiring authenticated access and other sensitive information	Utilize TLS (or its equivalent) for securing connections to any content that requires authenticated access or any other sensitive information.	When users access their email accounts, the connection should use TLS to protect the login credentials and email contents from interception.
9.5	Use TLS for connections to external systems involving sensitive information or functions	Employ TLS for securing connections to external systems that deal with sensitive information or critical functions.	When an application communicates with a third-party payment gateway to process financial transactions, it should use TLS to protect the data in transit.
9.6	Use a single standard TLS implementation that is configured appropriately	Implement a consistent and standard TLS configuration throughout the application to ensure proper and secure encryption.	Ensure that the application uses a well-established TLS implementation with the appropriate configuration settings to guarantee security.

9.7	Specify character encodings for all connections	Specify character encodings for all data transmitted over connections to avoid character encoding issues that can lead to security vulnerabilities.	When transmitting data between a web server and a database, specify UTF-8 character encoding to ensure compatibility and prevent encoding-related vulnerabilities.
9.8	Filter parameters containing sensitive information from the HTTP referer when linking to external sites	Exclude parameters containing sensitive information from the HTTP referer header when linking to external websites to prevent data leakage.	When a user clicks on an external link from an e-commerce website, the referer header should not include sensitive parameters like session tokens or personal data.

10. System Configuration

"System Configuration" is crucial for maintaining a secure environment for your application. This section includes practices like keeping servers and components up-to-date, minimizing privileges, removing unnecessary functionality, and securing HTTP methods. The main goal is to configure your systems in a way that minimizes vulnerabilities and protects against common attack vectors. Effective system configuration practices ensure that your application operates in a secure and robust environment, reducing the risk of security incidents and unauthorized access.

No	Test Case	Scenario	Example
10.1	Ensure servers, frameworks, and system components are running the latest approved version	Regularly check and update servers, frameworks, and system components to the latest approved versions to mitigate vulnerabilities.	An organization should regularly update its web server software to the latest approved version to patch known security vulnerabilities.
10.2	Ensure servers, frameworks, and system components have all patches issued for the version in use	Apply all security patches and updates issued for the specific version of servers, frameworks, and system components in use to address known vulnerabilities.	After deploying a web application using a specific framework version, apply all available patches and updates released for that version to keep it secure.
10.3	Turn off directory listings	Disable directory listings to prevent exposing sensitive information about the web server's directory structure to potential attackers.	When a web server receives a request for a directory that doesn't contain a default document (e.g., index.html), it should return a "403 Forbidden" error instead of listing the directory's contents.

10.4	Restrict the web server, process, and service accounts to the least privileges possible	Limit the permissions and access rights of web server, process, and service accounts to only what is necessary to perform their functions, reducing the risk of unauthorized access.	A web server process should run with minimal privileges, granting access only to the directories and resources required for serving web pages, and should not have write access to sensitive files.
10.5	When exceptions occur, fail securely	Implement error handling routines that ensure the application fails securely when exceptions or errors occur, preventing the exposure of sensitive information.	If a web application encounters an unexpected exception, it should handle it gracefully by displaying a user-friendly error message instead of revealing technical details or sensitive data.
10.6	Remove all unnecessary functionality and files	Eliminate any features, functions, or files that are not essential for the application's operation, reducing the attack surface.	An e-commerce website should remove any unused or unnecessary features, such as old product listings or deprecated functions, before deployment.
10.7	Remove test code or any functionality not intended for production, prior to deployment	Ensure that test code or features not intended for production use are removed from the application before it is deployed.	Any test-related functionality in the application's codebase should be excluded from the production release, preventing unintended exposure or vulnerabilities.

10.8	Prevent disclosure of your directory structure in the robots.txt file	Avoid exposing directory structures by isolating directories not intended for public indexing in an isolated parent directory and disallowing the entire parent directory in the robots.txt file.	If a website has directories containing configuration files or other sensitive data, these directories should be placed within a parent directory that is disallowed in the robots.txt file to prevent search engine indexing.
10.9	Define which HTTP methods (GET or POST) the application will support and whether they will be handled differently	Clearly specify which HTTP methods (e.g., GET or POST) the application supports and whether they are handled differently on different pages.	In a RESTful web service, define which HTTP methods are allowed for various endpoints, indicating whether a particular endpoint supports only GET requests or both GET and POST requests.
10.10	Disable unnecessary HTTP methods, such as WebDAV extensions	Deactivate any unnecessary HTTP methods, especially extensions like WebDAV, and only use well-vetted authentication mechanisms if required.	If a web application doesn't require the WebDAV HTTP extension for file management, it should disable this method to reduce potential security risks.

10.11	Ensure the web server handles HTTP 1.0 and 1.1 in a similar manner or understands any differences	Ensure that the web server is configured to handle both HTTP 1.0 and 1.1 consistently or understands any distinctions between the two versions.	If a web server accepts both HTTP 1.0 and HTTP 1.1 requests, it should process them in a consistent manner to prevent potential vulnerabilities related to version handling.
10.12	Remove unnecessary information from HTTP response headers related to the OS, web server version, and application frameworks	Minimize the information revealed in HTTP response headers to avoid disclosing details about the server's operating system, web server version, or application frameworks.	HTTP response headers should not expose server details, such as "Server: Apache/2.4.29 (Unix) PHP/7.2.15" or "X-Powered-By: Express."
10.13	The security configuration store for the application should be output in human-readable form to support auditing	Ensure that the security configuration settings of the application can be displayed in a human-readable format to facilitate auditing and review.	A web application's security settings should be documented in a way that allows auditors to easily understand the configurations for verification.

10.14	Implement an asset management system and register system components and software	Establish an asset management system to catalog and register all system components and software to monitor and maintain them efficiently.	An organization should use an asset management system to keep an inventory of all servers, network devices, software, and hardware components used in the infrastructure.
10.15	Isolate development environments from the production network and provide access only to authorized development and test groups	Segregate development environments from the production network, restricting access to authorized development and test groups.	Development and testing environments should be isolated from the live production network to minimize potential security risks and unauthorized access.
10.16	Implement a software change control system to manage and record changes to the code	Employ a software change control system to oversee and document changes made to the code, both in development and production.	Any modifications or updates to the application's source code should be documented in a change control system, including details like who made the change, when it was made, and why.

11. Database Security

"Database Security" is essential for ensuring the confidentiality and integrity of your application's data. This section includes practices like using strongly typed parameterized queries, input validation, and output encoding to prevent SQL injection attacks. It also emphasizes the importance of utilizing secure credentials for database access, storing connection strings securely, and minimizing privileges when interacting with the database. The primary goal is to protect your database from unauthorized access and data breaches, ensuring that sensitive information remains secure and confidential. Effective database security practices help safeguard your application's most critical asset – its data.

No	Test Case	Scenario	Example
11.1	Use strongly typed parameterized queries	Employ parameterized queries with strongly typed parameters to interact with the database, preventing SQL injection attacks.	When querying a database for user authentication, use parameterized queries with strongly typed parameters like integers or strings to avoid SQL injection.
11.2	Utilize input validation and output encoding and be sure to address meta characters. If these fail, do not run the database command	Apply input validation and output encoding to sanitize data, ensuring that meta characters are addressed to protect against security vulnerabilities. If validation fails, reject the database command.	When processing user-generated input for a search query, validate the input and ensure that it doesn't contain any unescaped meta characters such as single quotes. If validation fails, reject the query.

11.3	Ensure that variables are strongly typed	Ensure that variables used in database operations are strongly typed to prevent type-related vulnerabilities.	When passing variables to a database query, ensure that their data types match the expected data types in the database schema to avoid type conversion issues or unexpected behavior.
11.4	The application should use the lowest possible level of privilege when accessing the database	Limit the privileges granted to the application when interacting with the database to minimize the potential impact of security breaches.	When connecting to the database, use a database user account with the minimum necessary privileges to perform the required operations, rather than a superuser account.
11.5	Use secure credentials for database access	Implement strong and secure credentials when connecting to the database to protect against unauthorized access.	Use complex and unique passwords for database user accounts, and consider implementing multi-factor authentication for added security.

11.6	Connection strings should not be hard coded within the application. Connection strings should be stored in a separate configuration file on a trusted system and they should be encrypted.	Avoid hardcoding database connection strings in the application code. Store connection strings in a separate, encrypted configuration file on a trusted system.	Instead of directly embedding database connection strings in the code, store them in an encrypted configuration file external to the application for better security.
11.7	Use stored procedures to abstract data access and allow for the removal of permissions to the base tables in the database	Utilize stored procedures to access data, providing an abstraction layer and enabling the removal of permissions to base tables.	Implement stored procedures in the database to abstract data access, ensuring that applications interact with the data through these procedures.
11.8	Close the connection as soon as possible	Close the database connection as soon as it is no longer needed to reduce the risk of unauthorized access or data exposure.	After executing a database query, promptly close the database connection to minimize the window of opportunity for potential attackers.

11.9	Remove or change all default database administrative passwords. Utilize strong passwords/phrases or implement multi-factor authentication	Eliminate or update default administrative passwords for the database, replacing them with strong passwords or implementing multi-factor authentication for added security.	When deploying a new database, change the default administrative passwords to strong, unique passwords, or implement multi-factor authentication to enhance access security.
11.10	Turn off all unnecessary database functionality (e.g., unnecessary stored procedures or services, utility packages, install only the minimum set of features and options required)	Disable unnecessary database functionality and features, such as stored procedures or services, to reduce the attack surface.	When configuring a database server, only enable the features and options required for the application's functionality, turning off or uninstalling unnecessary components.
11.11	Remove unnecessary default vendor content (e.g., sample schemas)	Eliminate unnecessary default content provided by the database vendor, such as sample schemas or data, which can pose security risks.	After installing a database system, remove sample schemas, tables, or data provided by the vendor, as they might contain security vulnerabilities or unnecessary data.

11.12	Disable any default accounts that are not required to support business requirements	Deactivate or disable any default user accounts that are not needed to fulfill business requirements.	If the database system includes default user accounts that are not essential for the application's functionality, disable or remove them to reduce potential security risks.
11.13	The application should connect to the database with different credentials for every trust distinction (e.g., user, read-only user, guest, administrators)	Use distinct credentials for database connections based on trust levels, such as separate credentials for regular users, read-only access, guests, and administrators, to limit privileges as necessary.	Implement role-based database access, providing different user roles with unique credentials and access rights to match their trust levels, ensuring that administrators have more privileges than regular users.

12. File Management

"File Management" is all about handling files securely within your application. This section includes practices like not passing user-supplied data directly to dynamic include functions, authenticating file uploads, validating file types, and avoiding saving files in the same web context as the application. The primary goal is to prevent malicious file uploads, limit access to files, and ensure that files are handled safely to avoid security vulnerabilities. Effective file management practices help maintain the integrity and security of your application's file system and protect it from potential threats.

No	Test Case	Scenario	Example
12.1	Do not pass user supplied data directly to any dynamic include function	Avoid directly passing user-supplied data to dynamic include functions to prevent code execution vulnerabilities.	Instead of using user input to dynamically include a file, use a predefined and validated list of files to include.
12.2	Require authentication before allowing a file to be uploaded	Ensure that users are authenticated before they are allowed to upload files to the application to prevent unauthorized file uploads.	Only authenticated users should have the privilege to upload files.
12.3	Limit the type of files that can be uploaded to only those types that are needed for business purposes	Allow only specific types of files to be uploaded that are relevant to the application's business requirements.	If your application only requires image uploads, restrict file uploads to image file types (e.g., JPEG, PNG) and reject other file types.

12.4	Validate uploaded files are the expected type by checking file headers. Checking for file type by extension alone is not sufficient	Verify the file type of uploaded files by examining their headers to ensure they match the expected format, as relying solely on file extensions is insufficient.	When processing an uploaded file, check its header information to confirm its actual format rather than solely relying on the file extension provided by the user.
12.5	Do not save files in the same web context as the application. Files should either go to the content server or in the database.	Store uploaded files in a location separate from the web application's context to avoid security risks. Uploaded files should be placed in a content server or database.	Store uploaded files in a directory outside the web server's root directory to prevent direct access from the web.
12.6	Prevent or restrict the uploading of any file that may be interpreted by the web server.	Avoid uploading files that can be interpreted as code by the web server, as this may introduce security vulnerabilities.	Do not allow users to upload files like PHP, HTML, or JavaScript files that could be executed by the server.
12.7	Turn off execution privileges on file upload directories	Disable execution privileges on directories where files are uploaded to prevent the execution of uploaded files.	Modify directory permissions to prevent any uploaded files from being executed by the web server.

12.8	Implement safe uploading in UNIX by mounting the targeted file directory as a logical drive using the associated path or the chrooted environment	Secure file uploading in UNIX environments by mounting the target directory as a logical drive or using the chrooted environment to isolate the uploaded files.	When dealing with file uploads on UNIX systems, ensure that the uploaded files are placed in an isolated directory or chroot environment to enhance security.
12.9	When referencing existing files, use a white list of allowed file names and types. Validate the value of the parameter being passed and if it does not match one of the expected values, either reject it or use a hard coded default file value for the content instead	Employ a white list of permitted file names and types when referencing existing files, validating the parameters being passed. If an unexpected value is encountered, reject it or use a predefined default value.	When referencing files provided by users or external sources, validate the file names and types against a predefined white list, and only allow those that match the expected values.
12.10	Do not pass user supplied data into a dynamic redirect. If this must be allowed, then the redirect should accept only validated, relative path URLs	Avoid passing user-supplied data into dynamic redirects. If necessary, ensure that the redirect accepts only validated, relative path URLs to prevent open redirect vulnerabilities.	If your application allows dynamic redirects, ensure that the redirect URLs are either predefined and validated or accept only relative path URLs, reducing the risk of open redirect attacks.

12.11	Do not pass directory or file paths, use index values mapped to pre-defined list of paths	Avoid passing directory or file paths as user input, and instead, use index values associated with a predefined list of paths to enhance security.	When handling user input for file or directory paths, use predefined index values to reference specific paths rather than directly accepting arbitrary paths provided by users.
12.12	Never send the absolute file path to the client	Avoid disclosing absolute file paths to clients, as this information can potentially be exploited by attackers.	When returning file paths or URLs to the client, ensure that they are relative paths and do not reveal the absolute file system structure of the server.
12.13	Ensure application files and resources are read-only	Set appropriate file permissions to make application files and resources read-only, preventing unauthorized modification.	Configure file permissions on application files and resources to disallow write access, ensuring their integrity and preventing tampering.
12.14	Scan user uploaded files for viruses and malware	Implement virus and malware scanning for files uploaded by users to prevent malicious content from entering the application.	When users upload files, automatically scan the files for viruses and malware to safeguard the application and its users from potential threats.

13. Memory Management

"Memory Management" involves handling memory securely in your application. This section emphasizes practices like input and output control for untrusted data, checking buffer sizes to prevent buffer overflows, and avoiding known vulnerable functions. The goal is to ensure that your application efficiently manages memory, mitigates memory-related vulnerabilities, and avoids potential security risks associated with memory handling. Proper memory management contributes to the overall security and reliability of your software.

No	Test Case	Scenario	Example
13.1	Utilize input and output control for un-trusted data	Employ input and output controls to manage data from untrusted sources, ensuring that the data is processed safely and securely.	When accepting user input, apply input controls such as input validation to validate and sanitize the data before using it in the application. Similarly, use output controls like output encoding to prevent data from being executed as code when displayed to users.
13.2	Double check that the buffer is as large as specified	Verify that buffer sizes match the expected size to prevent buffer overflows and memory corruption vulnerabilities.	When using functions that copy data into a buffer, double-check that the destination buffer is of the specified size and that it can accommodate the data to be copied.

13.3	When using functions that accept a number of bytes to copy, such as strncpy(), be aware that if the destination buffer size is equal to the source buffer size, it may not NULL-terminate the string	Exercise caution when using functions like strncpy() and understand that if the destination buffer size is the same as the source buffer size, the string may not be NULL-terminated, which can lead to unexpected behavior.	When using strncpy(), be aware of its behavior and make sure to manually NULL-terminate the string if necessary to avoid issues with string manipulation.
13.4	Check buffer boundaries if calling the function in a loop and make sure there is no danger of writing past the allocated space	Ensure that buffer boundaries are validated when calling functions in loops, preventing potential buffer overflows and data corruption.	When using functions in loops that copy data into buffers, always check and control the loop's iteration to avoid writing data past the allocated buffer space.
13.5	Truncate all input strings to a reasonable length before passing them to the copy and concatenation functions	Limit the length of input strings to a reasonable size before using copy and concatenation functions to prevent buffer overflows.	Before using copy or concatenation functions, truncate input strings to a predefined reasonable length to ensure they fit within the buffer's allocated space.
13.6	Specifically close resources, don’t rely on garbage collection. (e.g., connection objects, file handles, etc.)	Explicitly close resources such as connection objects and file handles instead of relying on garbage collection to ensure timely resource deallocation.	When managing resources like database connections or file handles, always use explicit methods to close these resources once they are no longer needed to avoid resource leaks.

13.7	Use non-executable stacks when available	Employ non-executable stacks when possible to enhance security and prevent stack-based vulnerabilities.	If your operating system or platform supports non-executable stacks, enable this feature to reduce the risk of stack-based buffer overflow attacks.
13.8	Avoid the use of known vulnerable functions (e.g., printf, strcat, strcpy etc.)	Steer clear of using functions that are known to be vulnerable to security issues, such as printf, strcat, and strcpy.	Instead of using functions like printf, strcat, or strcpy, opt for safer alternatives that do not exhibit known vulnerabilities, such as printf-safe functions or string manipulation functions with boundary checks.
13.9	Properly free allocated memory upon the completion of functions and at all exit points	Ensure that dynamically allocated memory is correctly deallocated at the end of functions and at all exit points to prevent memory leaks.	When allocating memory dynamically (e.g., with malloc), always include code to release (free) the allocated memory in the function's exit paths to avoid memory leaks.

14. General Coding Practices

"General Coding Practices" encompass a set of guidelines for writing secure and reliable code. These practices encourage the use of approved managed code, task-specific APIs, and explicit variable initialization. They also emphasize avoiding direct interaction with the operating system and preventing concurrent access issues in multi-threaded applications. Additionally, the checklist promotes safe calculation handling, secure privilege management, and safeguarding against code injection and unsafe code alterations. Following these practices helps ensure that your code is robust, secure, and free from common vulnerabilities.

No	Test Case	Scenario	Example
14.1	Use tested and approved managed code rather than creating new unmanaged code for common tasks	Prefer using well-tested managed code libraries and APIs for common tasks over developing custom unmanaged code.	Instead of implementing custom unmanaged code for file I/O operations, use a widely accepted managed code library or API like .NET's File class for safe and efficient file operations.
14.2	Utilize task-specific built-in APIs to conduct operating system tasks. Do not allow the application to issue commands directly to the Operating System, especially through the use of application-initiated command shells	Employ built-in APIs specific to the task at hand to perform operating system-related actions. Avoid allowing the application to execute commands directly on the OS, especially through command shells initiated by the application.	When interacting with the operating system, use APIs like System.Diagnostics.Process in C# to manage and execute external processes instead of invoking shell commands through the application.

14.3	Use checksums or hashes to verify the integrity of interpreted code, libraries, executables, and configuration files	Verify the integrity of interpreted code, libraries, executables, and configuration files using checksums or cryptographic hashes to detect unauthorized modifications.	Before loading a dynamically linked library, verify its integrity by comparing its hash value with a precomputed hash to ensure it has not been tampered with.
14.4	Utilize locking to prevent multiple simultaneous requests or use a synchronization mechanism to prevent race conditions	Employ locking mechanisms to prevent multiple simultaneous requests or utilize synchronization to avoid race conditions in multi-threaded applications.	When managing shared resources, use locks or synchronization primitives such as mutexes to ensure that only one thread can access the shared resource at a time, preventing concurrent access issues.
14.5	Protect shared variables and resources from inappropriate concurrent access	Safeguard shared variables and resources to prevent inappropriate concurrent access that could lead to data corruption or inconsistency.	Protect shared data structures in a multi-threaded application by using proper locking mechanisms to ensure that multiple threads do not access or modify the data simultaneously.

14.6	Explicitly initialize all your variables and other data stores, either during declaration or just before the first usage	Ensure that all variables and data stores are explicitly initialized either during declaration or right before their first usage to prevent the use of uninitialized or unpredictable values.	Initialize variables with appropriate default values during declaration or initialize them just before their first use. For instance, initialize an integer variable with zero (0) to avoid using uninitialized data.
14.7	In cases where the application must run with elevated privileges, raise privileges as late as possible, and drop them as soon as possible	If the application needs to run with elevated privileges, elevate those privileges only when necessary and reduce them to the least privilege level as soon as they are no longer needed.	If your application requires elevated privileges to perform a specific task, raise those privileges only when executing that task, and promptly lower the privileges once the task is completed.

14.8	Avoid calculation errors by understanding your programming language's underlying representation and how it interacts with numeric calculation. Pay close attention to byte size discrepancies, precision, signed/unsigned distinctions, truncation, conversion and casting between types, "not-a-number" calculations, and how your language handles numbers that are too large or too small for its underlying representation	Prevent calculation errors by having a deep understanding of your programming language's numeric representation and how it handles various numeric operations and data types. Pay attention to issues like precision, signed/unsigned distinctions, byte size limitations, type conversion, and handling extreme values.	When performing mathematical calculations in a programming language, be aware of issues like integer overflow, floating-point precision, and type conversion. Always ensure that numeric operations are consistent with your expectations and the language's behavior.
14.9	Do not pass user-supplied data to any dynamic execution function	Avoid passing data provided by users to functions that dynamically execute code, as this can lead to code injection vulnerabilities.	Do not allow users to provide input that is directly passed to functions capable of executing code dynamically, such as eval() or dynamic SQL execution. This can prevent code injection attacks.

14.10	Restrict users from generating new code or altering existing code	Prevent users from generating or modifying code to maintain control over the application's behavior and security.	Avoid providing users with the capability to write or execute arbitrary code within the application, as this can lead to security risks.
14.11	Review all secondary applications, third-party code, and libraries to determine business necessity and validate safe functionality, as these can introduce new vulnerabilities	Conduct a thorough review of secondary applications, third-party code, and libraries to assess their necessity and ensure that their functionality does not introduce security vulnerabilities.	Before integrating third-party code or libraries into your application, carefully review and validate the code to confirm that it is both necessary for your business requirements and free from security vulnerabilities.

14.12	Implement safe updating. If the application will utilize automatic updates, then use cryptographic signatures for your code and ensure your download clients verify those signatures. Use encrypted channels to transfer the code from the host server	Implement secure update mechanisms, especially if your application supports automatic updates. Utilize cryptographic signatures to verify code authenticity, and ensure download clients validate these signatures. Additionally, use encrypted channels to transfer code updates from the host server.	When delivering automatic updates for your application, ensure that the updates are digitally signed with a cryptographic signature to guarantee their authenticity. The download client should verify these signatures before applying updates. Encrypt the communication channel between the host server and the client to protect the code updates from interception.

15. Glossary

- ★ **Abuse Case:** Describes the intentional and unintentional misuses of the software. Abuse cases should challenge the assumptions of the system design.
- ★ **Access Control:** A set of controls that grant or deny a user, or other entity, access to a system resource. This is usually based on hierarchical roles and individual privileges within a role, but also includes system to system interactions.
- ★ **Authentication:** A set of controls that are used to verify the identity of a user, or other entity, interacting with the software.
- ★ **Availability**: A measure of a system's accessibility and usability.
- ★ **Canonicalize:** To reduce various encodings and representations of data to a single simple form.
- ★ **Communication Security:** A set of controls that help ensure the software handles the sending and receiving of information in a secure manner.
- ★ **Confidentiality:** To ensure that information is disclosed only to authorized parties.
- ★ **Contextual Output Encoding:** Encoding output data based on how it will be utilized by the application. The specific methods vary depending on the way the output data is used. If the data is to be included in the response to the client, account for inclusion scenarios like: the body of an HTML document, an HTML attribute, within JavaScript, within a CSS or in a URL. You must also account for other use cases like SQL queries, XML and LDAP.
- ★ **Cross Site Request Forgery:** An external website or application forces a client to make an unintended request to another application that the client has an active session with. Applications are vulnerable when they use known, or predictable, URLs and parameters; and when the browser automatically transmits all required session information with each request to the vulnerable application. (This is one of the only attacks specifically discussed in this document and is only included because the associated vulnerability is very common and poorly understood.)
- ★ **Cryptographic Practices:** A set of controls that ensure cryptographic operations within the application are handled securely.
- ★ **Data Protection:** A set of controls that help ensure the software handles the storing of information in a secure manner.
- ★ **Database Security:** A set of controls that ensure that software interacts with a database in a secure manner and that the database is configured securely.

★ **Error Handling and Logging:** A set of practices that ensure the application handles errors safely and conducts proper event logging.

★ **Exploit:** To take advantage of a vulnerability. Typically this is an intentional action designed to compromise the software's security controls by leveraging a vulnerability.

★ **File Management:** A set of controls that cover the interaction between the code and other system files.

★ **General Coding Practices:** A set of controls that cover coding practices that do not fit easily into other categories.

★ **Hazardous Character:** Any character or encoded representation of a character that can affect the intended operation of the application or associated system by being interpreted to have a special meaning, outside the intended use of the character. These characters may be used to:

- Alter the structure of existing code or statements
- Insert new unintended code
- Alter paths
- Cause unexpected outcomes from program functions or routines
- Cause error conditions
- Have any of the above effects on downstream applications or systems

★ **HTML Entity Encode:** The process of replacing certain ASCII characters with their HTML entity equivalents. For example, encoding would replace the less-than character "<" with the HTML equivalent "<". HTML entities are 'inert' in most interpreters, especially browsers, which can mitigate certain client-side attacks.

★ **Impact:** A measure of the negative effect on the business that results from the occurrence of an undesired event; what would be the result of a vulnerability being exploited.

★ **Input Validation:** A set of controls that verify the properties of all input data match what is expected by the application, including types, lengths, ranges, acceptable character sets, and does not include known hazardous characters.

★ **Integrity:** The assurance that information is accurate, complete, and valid and has not been altered by an unauthorized action.

★ **Log Event Data:** This should include the following:

- Time stamp from a trusted system component
- Severity rating for each event
- Tagging of security-relevant events if they are mixed with other log entries
- Identity of the account/user that caused the event
- Source IP address associated with the request
- Event outcome (success or failure)
- Description of the event

★ **Memory Management:** A set of controls that address memory and buffer usage.

★ **Mitigate:** Steps taken to reduce the severity of a vulnerability. These can include removing a vulnerability, making a vulnerability more difficult to exploit, or reducing the negative impact of a successful exploitation.

★ **Multi-Factor Authentication:** An authentication process that requires the user to produce multiple distinct types of credentials. Typically, this is based on something they have (e.g., a smart card), something they know (e.g., a pin), or something they are (e.g., data from a biometric reader).

★ **Output Encoding:** A set of controls addressing the use of encoding to ensure data output by the application is safe.

★ **Parameterized Queries (prepared statements):** Keeps the query and data separate through the use of placeholders. The query structure is defined with placeholders, the SQL statement is sent to the database and prepared, and then the prepared statement is combined with the parameter values. This prevents the query from being altered because the parameter values are combined with the compiled statement, not a SQL string.

★ **Sanitize Data:** The process of making potentially harmful data safe through the use of data removal, replacement, encoding, or escaping of the characters.

★ **Security Controls:** An action that mitigates a potential vulnerability and helps ensure that the software behaves only in the expected manner.

★ **Security Requirements:** A set of design and functional requirements that help ensure the software is built and deployed in a secure manner.

★ **Sequential Authentication:** When authentication data is requested on successive pages rather than being requested all at once on a single page.

★ **Session Management:** A set of controls that help ensure web applications handle HTTP sessions in a secure manner.

★ **State Data:** When data or parameters are used by the application or server to emulate a persistent connection or track a client's status across a multi-request process or transaction.

★ **System:** A generic term covering the operating systems, web server, application frameworks, and related infrastructure.

★ **System Configuration:** A set of controls that help ensure the infrastructure components supporting the software are deployed securely.

★ **Threat Agent:** Any entity that may have a negative impact on the system. This may be a malicious user who wants to compromise the system's security controls; however, it could also be an accidental misuse of the system or a more physical threat like fire or flood.

★ **Trust Boundaries:** Typically, a trust boundary constitutes the components of the system under your direct control. All connections and data from systems outside of your direct control, including all clients and systems managed by other parties, should be considered untrusted and be validated at the boundary before allowing further system interaction.

★ **Vulnerability:** A weakness that makes the system susceptible to attack or damage.