

# **SECURE CODING PRACTICES CHECKLIST (OWASP)**

INPUT VALIDATION

OUTPUT ENCODING

AUTHENTICATION AND PASSWORD MANAGEMENT

SESSION MANAGEMENT

ACCESS CONTROL

CRYPTOGRAPHIC PRACTICES

ERROR HANDLING AND LOGGING

DATA PROTECTION

COMMUNICATION SECURITY

SYSTEM CONFIGURATION

DATABASE SECURITY

FILE MANAGEMENT

MEMORY MANAGEMENT

GENERAL CODING PRACTICES

# INPUT VALIDATION

---

- Perform input validation on a secure system (server-side, not client-side).
- Identify and categorize all data sources into trusted and untrusted.
- Validate all data from untrusted sources, such as databases and file streams.
- Use a centralized input validation mechanism for the entire application.
- Specify the character sets (e.g., UTF-8) for all input sources to ensure consistency (canonicalization).
- Encode inputs to a common character set before validation.
- Reject all inputs that fail validation.
- If the system supports UTF-8 extended character sets, validate inputs after decoding the UTF-8 encoding.
- Validate all client-provided data before processing it.
- Ensure protocol header values in both requests and responses contain only ASCII characters.
- Validate data from redirects to ensure integrity and security.
- Use an allow list to validate expected data types, rather than a deny list.
- Validate data ranges to ensure they fall within acceptable limits.
- Validate data lengths to ensure they conform to expected sizes.
- If potentially dangerous input must be accepted, implement additional security measures to mitigate risk.
- For inputs that the standard validation routine cannot fully address, use extra specific checks.
- Use canonicalization to handle obfuscation attacks and ensure inputs are processed consistently.

## OUTPUT ENCODING

---

- Perform all output encoding on a secure system (server-side, not client-side).
- Use a standardized, tested encoding routine for each type of outbound output.
- Specify character sets, such as UTF-8, for all outputs to ensure consistency.
- Contextually encode all data returned to the client from untrusted sources to prevent security vulnerabilities.
- Ensure the output encoding is safe and compatible with all target systems.
- Contextually sanitize all output of untrusted data before it is used in SQL, XML, and LDAP queries to prevent injection attacks.
- Sanitize all output of untrusted data before passing it to operating system commands to avoid command injection risks.

# AUTHENTICATION AND PASSWORD MANAGEMENT

---

- Require authentication for all resources except explicitly public ones.
- Use trusted systems for authentication controls, implementing centralized management.
- Utilize standardized, well-tested authentication services.
- Separate authentication logic from requested resources via redirection.
- Ensure authentication mechanisms fail securely.
- Hash passwords server-side with cryptographically strong, one-way salted hashes.
- Transmit passwords only over encrypted connections via HTTP POST.
- Enforce password complexity, length, and change policies per regulations.
- Prevent password reuse and require at least one-day-old passwords for changes.
- Mask password input and disable "remember me" functionality.
- Notify users Lock accounts after a defined number of failed login attempts.
- Secure password reset/change processes with equivalent security to authentication.
- Use email-based resets with temporary links or passwords and short expiration periods.
- Require immediate password changes for temporary credentials.
- Validate reset questions to prevent predictable answers.of password resets and last login activity.
- Implement Multi-Factor Authentication (MFA) for high-risk accounts.

- Re-authenticate users for critical operations.
- Secure credentials for external services in secure stores.
- Change default vendor credentials and monitor attacks on user accounts.
- Regularly inspect third-party authentication code for vulnerabilities.

## SESSION MANAGEMENT

---

- Use server-side session management controls exclusively.
- Create session identifiers on trusted systems using vetted algorithms for randomness.
- Avoid exposing session identifiers in URLs, error messages, or logs.
- Restrict cookie domain and path appropriately.
- Set cookies with Secure and HttpOnly attributes, unless client-side access is essential.
- Consistently use HTTPS for all communications.
- Terminate sessions fully on logout; provide logout functionality on all protected pages.
- Establish short session inactivity timeouts and enforce periodic session terminations.
- Disallow persistent logins and concurrent logins with the same user ID.
- Close pre-login sessions and establish new ones upon successful login.
- Generate new session identifiers after re-authentication, periodic intervals, or security context changes (e.g., HTTP to HTTPS).
- Use per-session strong random tokens for sensitive server-side operations.
- Employ per-request strong random tokens for critical actions like account management.
- Implement controls to protect server-side session data from unauthorized access by other users.

## ACCESS CONTROL

---

- Use only trusted server-side objects for access authorization decisions.
- Centralize authorization checks site-wide, including external service calls.
- Enforce authorization on every request, including server-side scripts.
- Ensure access controls fail securely and deny access if security configurations are unavailable.
- Restrict access to files, resources, URLs, functions, services, application data, and security configurations to authorized users only.
- Match server-side implementation with presentation layer access rules.
- Limit direct object references and state tampering using encryption and integrity checks.
- Segregate privileged logic from other application code.
- Use the least privilege principle for service accounts and external system connections.
- Periodically re-validate user authorization in long sessions and log users out if privileges change.
- Disable unused accounts and enforce session termination upon authorization revocation.
- Enforce application logic flows to align with business rules.
- Document access policies, including business rules, data types, and authorization criteria.
- Monitor and audit accounts, ensuring compliance with access requirements.
- Limit transaction rates to deter automated attacks while meeting business needs.

- Use the "referer" header as a supplemental, not primary, authorization check.

## CRYPTOGRAPHIC PRACTICES

---

- Perform all cryptographic operations on a trusted system.
- Protect secrets from unauthorized access and ensure cryptographic modules fail securely.
- Use approved random number generators for all random data (e.g., file names, GUIDs, strings).
- Utilize cryptographic modules compliant with FIPS 140-2 or equivalent standards.
- Establish policies and processes for cryptographic key management.

# ERROR HANDLING AND LOGGING

---

- Avoid disclosing sensitive information (e.g., system details, session identifiers, account data) in error responses.
- Use generic error messages with custom error pages.
- Implement application-level error handling rather than relying on server configurations.
- Free allocated memory during error conditions and deny access by default in security control errors.
- Implement logging on a trusted system and centralize all logging operations.
- Support logging for both success and failure of specified security events.
- Ensure logs contain critical event data without sensitive information (e.g., passwords, session identifiers).
- Restrict log access to authorized personnel and prevent untrusted data in logs from executing as code.
- Enable mechanisms for log analysis and validate log entry integrity using cryptographic hash functions.
- Input validation failures.
- All authentication attempts, especially failures.
- Access control failures and state tampering events.
- Attempts with invalid/expired session tokens.
- System exceptions and backend TLS connection failures.
- Cryptographic module failures.
- Administrative actions, including security configuration changes.

# DATA PROTECTION

---

- Enforce least privilege by restricting users to only the required functionality, data, and system information for their tasks.
- Protect cached or temporary sensitive data stored on the server, ensuring timely purging once no longer required.
- Encrypt highly sensitive stored information (e.g., authentication data), even on the server side.
- Safeguard server-side source code from unauthorized downloads.
- Avoid storing passwords, connection strings, or other sensitive data in clear text or insecurely on the client side.
- Remove comments from production code and unnecessary system documentation that might expose backend details or sensitive information.
- Do not include sensitive data in HTTP GET request parameters.
- Disable auto-complete for sensitive fields, such as those containing authentication credentials.
- Prevent client-side caching on pages displaying sensitive data.
- Ensure the application supports data removal once it is no longer required.
- Apply strict access controls to sensitive data on the server, including cached and temporary files, limiting access to authorized users only.
- Encrypt all sensitive data transmissions using TLS.
  - Include TLS for authenticated content and sensitive files.
  - Ensure TLS certificates are valid, match the domain name, are not expired, and include intermediate certificates if necessary.
- Prevent fallback to insecure connections after failed TLS attempts.

- Use TLS for external system connections involving sensitive information or functions.
- Standardize on a single, appropriately configured TLS implementation.
- Specify character encodings for all connections.
- Filter sensitive parameters from the HTTP referer header when linking to external sites.

## SYSTEM CONFIGURATION

---

- Ensure servers, frameworks, and components are running the latest approved versions.
- Apply all relevant patches to the versions in use.
- Restrict web server, process, and service accounts to least privilege.
- Turn off directory listings to prevent unauthorized browsing.
- Disable unnecessary HTTP methods and ensure supported methods (e.g., GET or POST) are handled securely and consistently across application pages.
- Configure all supported HTTP versions similarly and document any differences.
- Remove unnecessary information from HTTP response headers (e.g., OS, web-server version, application frameworks).
- Prevent directory structure exposure in the robots.txt file by isolating directories not intended for public indexing.
- On exceptions, ensure the application and server fail securely.
- Remove unnecessary functionality, test code, or files not intended for production before deployment.
- Isolate development environments from production networks, restricting access to authorized personnel.
- Implement a software change control system to manage and log code changes in development and production environments.
- Ensure the security configuration store is human-readable to support auditing.

- Establish an asset management system to track and register all system components and software.

# DATABASE SECURITY

---

- Use strongly typed parameterized queries to prevent SQL injection.
- Implement input validation and output encoding to handle meta characters securely. If validation fails, do not execute the database command.
- Ensure all variables are strongly typed.
- Access the database with the lowest possible privilege level required for operations.
- Use secure credentials for database access.
- Store connection strings in an encrypted, separate configuration file on a trusted system; do not hard-code them in the application.
- Assign different credentials for each trust distinction (e.g., users, read-only users, guests, administrators).
- Remove or change all default administrative passwords.
- Turn off unnecessary database functionality.
- Remove unnecessary default vendor content, such as sample schemas.
- Disable any default accounts that are not required for business needs.
- Use stored procedures to abstract data access and remove direct permissions to base tables.
- Close database connections as soon as possible to reduce exposure.

## FILE MANAGEMENT

---

- Do not pass user supplied data directly to any dynamic include function
- Require authentication before allowing a file to be uploaded
- Limit the type of files that can be uploaded to only those types that are needed for business purposes
- Validate uploaded files are the expected type by checking file headers rather than by file extension
- Do not save files in the same web context as the application
- Prevent or restrict the uploading of any file that may be interpreted by the web server.
- Turn off execution privileges on file upload directories
- Implement safe uploading in UNIX by mounting the targeted file directory as a logical drive using the associated path or the chrooted environment
- When referencing existing files, use an allow-list of allowed file names and types
- Do not pass user supplied data into a dynamic redirect
- Do not pass directory or file paths, use index values mapped to pre-defined list of paths
- Never send the absolute file path to the client
- Ensure application files and resources are read-only
- Scan user uploaded files for viruses and malware

# MEMORY MANAGEMENT

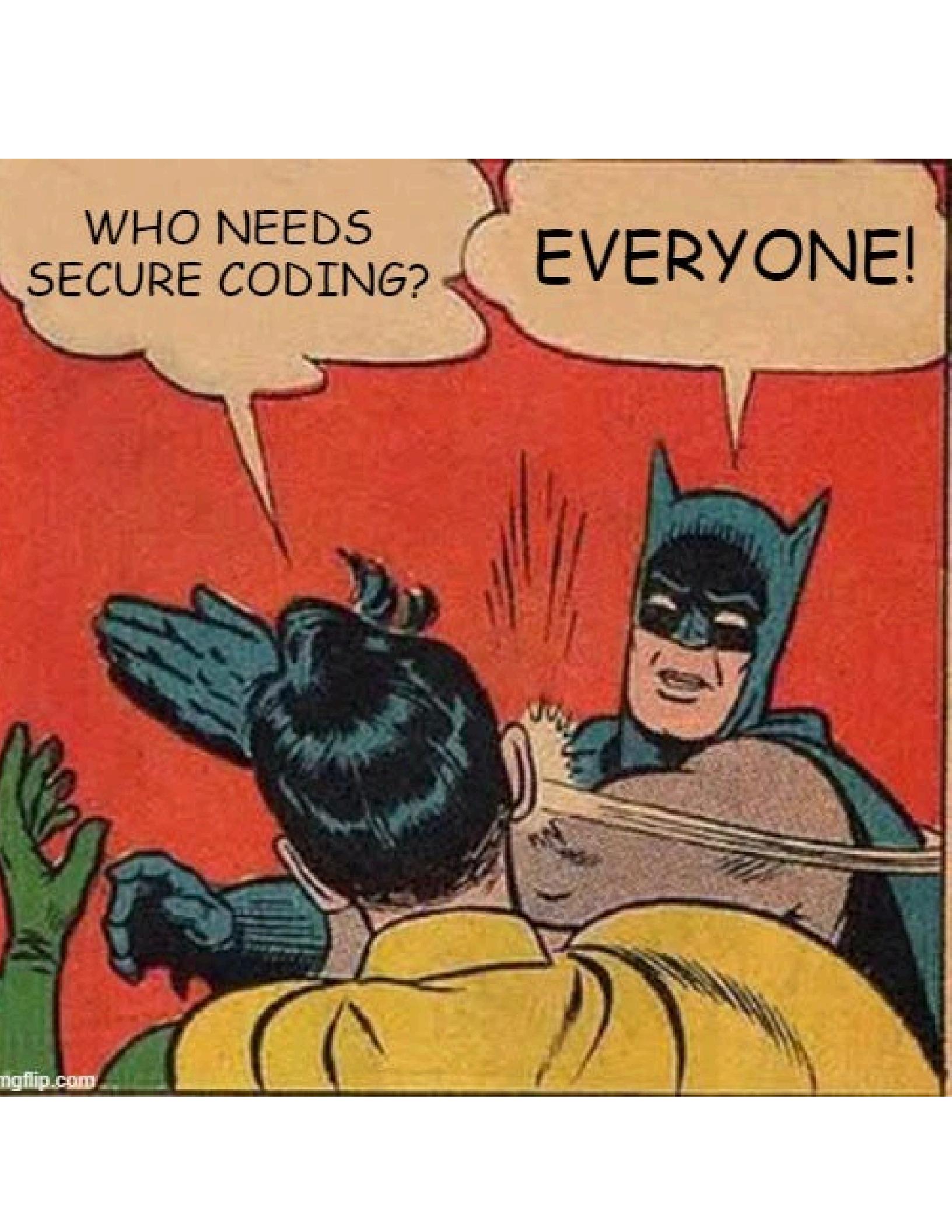
---

- Utilize input and output controls for untrusted data
- Check that the buffer is as large as specified
- When using functions that accept a number of bytes ensure that NULL termination is handled correctly
- Check buffer boundaries if calling the function in a loop and protect against overflow
- Truncate all input strings to a reasonable length before passing them to other functions
- Specifically close resources, don't rely on garbage collection
- Use non-executable stacks when available
- Avoid the use of known vulnerable functions
- Properly free allocated memory upon the completion of functions and at all exit points
- Overwrite any sensitive information stored in allocated memory at all exit points from the function

## GENERAL CODING PRACTICES

---

- Use tested and approved managed code instead of creating unmanaged code for common tasks.
- Leverage task-specific built-in APIs for OS interactions; avoid issuing direct OS commands or using application-initiated command shells.
- Review secondary applications, third-party code, and libraries for business necessity and safe functionality.
- Use checksums or hashes to verify the integrity of interpreted code, libraries, executables, and configuration files.
- Implement secure updating mechanisms using encrypted channels.
- Use locking mechanisms or synchronization to prevent race conditions.
- Protect shared variables and resources from inappropriate concurrent access.
- Explicitly initialize all variables during declaration or before first use.
- When elevated privileges are necessary, raise them as late as possible and drop them immediately after use.
- Avoid calculation errors by understanding the underlying representation of your programming language.
- Do not pass user-supplied data to dynamic execution functions.
- Restrict users from generating or altering code.
- Validate that all secondary applications, third-party code, and libraries are necessary and safe.



WHO NEEDS  
SECURE CODING?

EVERYONE!