DP WORLD

Secure Coding Practices and Guidelines

1. Document Management Information

Prepared By:

Item	Description				
Document Title	Secure Coding Practices and Guidelines				
Document Ref	Information Security Guidelines				
Document ID	DPW_IS_GL003		**Version**		1.0
Classification	○ Public	[✓ Internal]	○ Confidential	○ Strictly Confidential	○ Classified (Secret)
Status	Published	**Type**	Doc	**Owner / Department**	Global IT
Publish Date					

Version Control:

Version	Date	Author(s)	Change Description
0.1	03/03/2019	Muhammad Tariq Ali	Draft version
1.0	03/03/2019	Muhammad Tariq Ali	Final Version

Reviewers:

Version	Date	Reviewer(s)	Title	Remark
1.0	03/03/2019	Sajjad Ahmad	Manager – Group IT Governance	Reviewed

Approvers:

Version	Date	Approver(s)	Title	Remark

Contents

2. Introduction

The following sections define a set of general security coding practices that can be integrated in the software development lifecycle. Implementation of these practices will mitigate most common software vulnerabilities including OWASP Top 10. This provides coding practices that can be translated into coding requirements without the need for the developer to have an in depth understanding of security vulnerabilities and exploits.

The goal of software security is to maintain the confidentiality, integrity, and availability of information resources in order to enable successful business operations. This goal is accomplished through the implementation of security controls. The section below focuses on the technical controls specific to mitigating the occurrence of common software vulnerabilities.

2.1. Input Validation and Output Validation

Input validation is the process of validating inputs received from user, subsystem, component or an external entity (Web services). Each input received should never be trusted and should be validated for business value and malicious input (such as script, OS/DB commands and so on). Input validation is the best practice to countermeasure against the attacks such as Cross-site scripting (XSS), Sql Injection, Command injection and so on.

- Prepare a list of inputs that the software will process, such as user input, text file, feed and so on, and list the types of input that are not validated (with rationale). Identify all the potential interfaces through which software will receive input data. List can include but not limited to URL components, request headers, parameters, arguments, cookies, file, socket, environment variables, e-mail, and databases. Perform input validation at well-defined interfaces.
- Assume that all inputs are malicious until proven otherwise. As a minimum, validate input for data type, length, range, format, syntax, missing or extra inputs, meta characters, consistency across related fields and conformance to business rules. Use 'white' list of allowed characters, whenever possible for validating or sanitising all input.
- Deploy common centralised library and ensure that all developers use it for validation, filtering and sanitisation. Deploy same validations at client and server side. While making a call to external programmes such as library, executable or native code, validate inputs to ensure that inputs sent to them are not violating any of expectations.
- If application receives data from multiple sources for any operation, validate individual data while receiving from sources. If data to be combined with each other for operation then first combine all required data and perform the validation.
- Use appropriate encoding such as ISO 8859-1 or UTF 8 for each interface including output. While exchanging data amongst components, ensure that all components use same character encoding. If protocol permits then explicitly set character encoding. For web applications, use HTML encoding and URL encoding for user input when writing back to the client.
- If your standard validation routine cannot address the following inputs, then they should be checked discretely
  - Check for null bytes (%00)
  - Check for new line characters (%0d, %0a, \r, \n)

- ➢ Check for "dot-dot-slash" (../ or ...\) path alterations characters. In cases where UTF-8 extended character set encoding is supported, address alternate representation such as: %c0%ae%c0%ae/
- If any potentially hazardous characters must be allowed as input, be sure that you implement additional controls such as output encoding, secure task specific APIs and accounting for the utilization of that data throughout the application. Examples of common hazardous characters include: < > " ' % ( ) & + \ \' \"

2.2. Authentication

Authentication is the process of validating a user's identity before allowing access to resources. It is required whenever trust boundary is crossed. Traditionally, authentication is implemented through 'Something you have' (such as a key, a token, a smartcard), "something you know" (such as account details, PIN, password), or "something you are" (such as fingerprints, retina image) or even a combination of them. An authentication technique that employs such a combination is known as multifactor authentication.

- Identify resources that require authentication. Reduce the attack surface by distributing the software functionality into three parts – anonymous, registered users, and privileged users. Verify authenticity of source before processing any requests.
- Depending on business requirements, enforce password complexity, password renewal (to prevent aging), and old password reuse prevention. Ensure that privileged accounts have stringent authentication requirements.
- The software must employ an account with the least privileges towards accessing back-end systems such as file system, database, web service, and message queue. Re-authenticate user accounts in case of privileged operations and sensitive transactions.
- Protect the authentication token by using encryption and secure communication channels. Further, ensure that the token expires after a pre-determined period of time.
- Depending on the business requirements, temporarily or permanently disable the user account or IP address after a specified number of unsuccessful attempts. If more than one account is disabled following requests from the same IP address in a short time, generate an alert and block that IP address.
- Authentication failure responses should not indicate which part of the authentication data was incorrect. For example, instead of "Invalid username" or "Invalid password", just use "Invalid username and/or password" for both. Error responses must be truly identical in both display and source code.
- Authentication credentials for accessing services external to the application should be encrypted and stored in a protected location on a trusted system (e.g. server). The source code is NOT a secure location.
- Use a centralised implementation for all authentication controls, including libraries that call external authentication services. Segregate authentication logic from the resource being requested and use redirection to and from the centralised authentication control.

2.3. Authorization

Authorization is the process of controlling access and rights to resources such as file, data, configuration, program, and service for authenticated identities. A web application should protect front-end and back-end data and system resources by implementing access control restrictions on what users can do, which resources they have access to, and what functions they are allowed to perform on the data.

Generally, authorization is carried out after authentication and is imposed on the basis of policies or rules. Improper or weak authorization leads to information disclosure, abuse of functionality, and data tampering etc.

- Validate each request for authorization of web applications; ensure authorization is checked before each page access with mechanism such as server-side session token.
- Offer minimal functionality to anonymous users (if possible, deny all access). Provide no privileges to an account when it is first created and grant permissions incrementally.
- Do not rely on client data (such as URL parameter, Hidden Parameter, POST parameter, Cookie parameter and so on). Store users related information in session during login and only use the session variable each time for granting the access.
- Avoid storing sensitive information such as USER_ID, ROLE_ID and so on in cookies; instead use session variables to store this information on the server side.
- Do not rely on JavaScript functions or URLs and implement strong server-side validation and authorisation based on access control list (ACL).
- If necessary, implement directory specific access control by using the configuration file settings.
- Remove all the unused files such as old backup, unused PDF documents\spread sheets\reports, old source code files, default server manuals and documentation, temporary files, log files, press releases and so on from the production server. If you want to store these files, keep them separately in a folder\directory which is outside the Web\application servers' installation directory so that these documents are not accessible to any user.
- Implement account auditing and enforce the disabling of unused accounts (e.g., after no more than 30 days from the expiration of an account's password.)

2.4. Audit Trial and Logging

An audit trail is a chronological sequence of audit records, where each record comprises evidence of the actions performed, the location from which the actions were performed, and the entities that performed those actions. It also assists in adhering to compliance regulations, intrusion detection, incident investigation, software trouble shooting, and program debugging.

- Ensure that server time is synchronized with NTP, other applications and systems. Ensure that all the records in the audit trail contain the timestamps, IP addresses, action owners, events and descriptions etc.
- Ensure that log files are stored in a safe location of the file system and that adequate disk space is available for logging and carry out frequent backup of logs.

- Store log files on a separate partition and provide restricted privileges to log file access to avoid the possibility of illegal alteration, display, or deletion.
- Logging controls should support both success and failure of specified security events.
- Do not store sensitive information in logs, including unnecessary system details, session identifiers or passwords.
- Log all input validation failures
- Log all apparent tampering events, including unexpected changes to state data
- Log attempts to connect with invalid or expired session tokens
- Log all system exceptions
- Log all administrative functions, including changes to the security configuration settings
- Use a cryptographic hash function to periodically validate log entry integrity

2.5. Session Management

For multi user systems, users operate independently, and it is important to maintain their session securely. Stateless protocol such as HTTP provides no means for the server to maintain user state. The server deploys a session – a server-side mechanism to provide continuous session-like experience to users. When a user sends the first request, the server generates a unique session identifier (SID) and sends it to the client. The client then sends the SID back along with each subsequent request. The SID functions as an identification token for users, and servers use the SID to maintain session data.

- Identify all locations where user state is maintained, analyse attacker’s accessibility and build control around them. Do not allow the user to modify state information directly; state changes shall be allowed through legitimate actions only.
- Generate the session identifier (SID) with appropriate randomness and length using the framework's session manager rather than creating a weak SID using a home-grown algorithm.
- Use secure communication channels (such as SSL/TLS) to secure SID. Renew the SID before a sensitive transaction.
- Logout functionality should be available from all pages protected by authorisation. When a user logs out, invalidate all the associated connection and session and do not reuse the same SID for another user, and remove the SID from server.
- Disallow persistent logins and enforce periodic session terminations, even when the session is active. Especially for applications connecting to critical systems. Termination times should support business requirements and the user should receive sufficient notification to mitigate negative impacts
- Establish a session inactivity timeout that is as short as possible, based on balancing risk and business functional requirements.
- Generate a new session identifier on any re-authentication
- Generate a new session identifier and deactivate the old one periodically. This can mitigate certain session hijacking scenarios where the original identifier was compromised

- Do not expose session identifiers in URLs, error messages or logs. Session identifiers should only be located in the HTTP cookie header. For example, do not pass session identifiers as GET parameters.
- Set the domain and path for cookies containing authenticated session identifiers to an appropriately restricted value for the site.
- Generate a new session identifier if the connection security changes from HTTP to HTTPS, as can occur during authentication. However, it is recommended to consistently utilize HTTPS rather than switching between HTTP to HTTPS.
- Use a one-time unique identifier as the hidden parameter in the form (for example, if the same user downloads the form twice, the value of the unique identifiers will be different) and remove it from the active list as soon as the form is submitted. Ensure identifier is not predictable.
- Supplement standard session management for sensitive server-side operations, such as account management, by utilizing per-session strong random tokens or parameters. This method can be used to prevent Cross Site Request Forgery (CSRF) attacks.
- Set cookies with the HttpOnly attribute, unless you specifically require client-side scripts within your application to read or set a cookie's value
- Set the "secure" attribute for cookies transmitted over an TLS connection.

2.6. Data Protection

Data security is the means of ensuring that data is kept safe from corruption, unauthorized access, and disclosure, and thus helps ensure privacy. Whenever data is created, acquired, processed, transferred, stored, altered, retrieved, or destroyed, it must follow a secure path.

- Implement least privilege; restrict users to only the functionality, data and system information that are required to perform their tasks.
- Protect all cached or temporary copies of sensitive data stored on the server from unauthorised access and purge those temporary working files a soon as they are no longer required.
- Do not hardcode secret (such as password, key, database or application server connection information) or private information into the code; rather store them into configuration files into encrypted format. Use secure communication channels (such as SSL/TLS) to secure data in transit.
- Encrypt highly sensitive stored information, such as authentication verification data, even on the server side. Always use well vetted algorithms for encryption.
- Do not store passwords, connection strings or other sensitive information in clear text or in any non-cryptographically secure manner on the client side.
- Remove comments in user accessible production code that may reveal backend system or other sensitive information.
- Disable auto complete features on forms expected to contain sensitive information, including authentication.

- Disable client-side caching on pages containing sensitive information. Cache-Control: no-store, no cache, must revalidate may be used in conjunction with the HTTP header control "Pragma: no-cache"
- Collect end user data only if absolutely necessary and relevant in given context. Provide appropriate notice to user about what data is being collected, for what purpose it is collected, retained, stored and how it will be used. Provide option to user to deny data collection. If it is continuous collection, take user’s consent before starting data collection with defined period.

2.7. Cryptography

Cryptography is the process of hiding information. Traditionally, it is used to protect confidentiality and integrity of sensitive data. However, it can also be used for authentication and non-repudiation (through digital certificate). Cryptography can be implemented using symmetric (involved parties share a common key) or asymmetric (two inter-dependent keys are generated; one key is used for encryption and the other is used for decryption) mechanisms or a combination of both depending on the requirement.

- Avoid using home-grown cryptographic algorithms; rather, only use publicly scrutinized cryptographic algorithms such as AES, RSA, and DSA with recommended key sizes.
- Do not hardcode keys into source code. Use secure storage (such as JKS, PCKS12) to store the keys on disk. If need arise to put keys on disk, set access control to protect it against unauthorised disclosure, alteration, or deletion.
- Only use cryptographically strong PRNG to generate keys and session identifiers. Platform-specific APIs include: For java, use Secure Random.
- Configure server-side certificate to support SSLv3/TLSv1 or above and do not support weak versions such as SSLv2.
- Use standard protocols to establish trust (such as SSL/TLS and Kerberos) while exchanging keys.
- Ensure the server's certificate has a valid future date, that the issued name matches the expected hostname, and that the certificate is endorsed by a trusted root certification authority (CA).
- Remove the certificate import file from the disk once the certificate is imported into the key store. (Encryption is pointless, if the certificate or key is left unprotected on the same server).

Recommended algorithms and protocols are given in following tables for developer references.

Algorithm	Is Recommended?	Key Size (Bits)
AES	Yes	128 minimum, 256+ advised
DHE	Yes	1024 minimum
ECDHE	Yes	256 minimum 384 advised
DES,3DES	No	128 minimum
RSA	Yes	1024 minimum, 2048 advised
DSA	Yes	128 minimum, 256+ advised
MD5	No	NA
SHA-1	Yes	Only if mandated for compatibility and SHA-2 and above are not usable
SHA-2	Yes	SHA-256 and above
SHA-3	Yes	SHA-256 and above

Cryptography Protocol

Protocol	Is Recommended?
SSLv2	No
SSLv3	No
TLS 1.0	No
TLS 1.1	Yes
TLS 1.2	Yes
SSHv1	No
SSHv2	Yes
Kerberosv4	Yes
Kerberosv5	Yes

2.8. Error and Exception Handling

Error and exception handling bring reliability to the program and assure that it fails safely under all possible error conditions. Failure to handle erroneous conditions may lead the program to an insecure state that usually results in information disclosure. For example, if an attacker can force exceptions to occur and you fail to correctly handle these situations you will expose sensitive information about the inner workings of the application. These detailed error messages will help attackers build a picture of your application and fine tune their attacks. An attack such as SQL Injection will become significantly easy to exploit if an attacker can view the database information through internal error messages.

- Do not use static error strings in the code; instead, use configuration/property files. Construct an "Error code book" for unique error messages with error code for error logging. Map such messages to generic error messages for end user viewing.
- Production code should not be capable of producing debug messages or comments for developer. In the event of a failure, do not expose information (such as verbose messages, stack traces, or path details) that could lead to information disclosure.
- If possible deploy centralised exception management strategy and ensure that all developers share a common approach to exception handling.
- All resource allocation must be handled with the aid of appropriate exception. Do not incorporate an empty try/catch block into the code.
- Do not handle all exceptions with a single try/catch/finally block; rather, handle exception near the source. Log all errors/exceptions raised in sensitive functions.
- Exceptions that are thrown while logging is in progress can prevent successful logging. Failure to account for exceptions during the logging process can cause security vulnerabilities, such as allowing an attacker to conceal critical security exceptions by preventing them from being logged. Therefore, programs must ensure that data logging continues to operate correctly even when exceptions are thrown during the logging process.
- Do not throw RuntimeException, Exception that is, throwing a RuntimeException can lead to subtle errors; for example, a caller cannot examine the exception to determine why it was thrown and consequently cannot attempt recovery. However, methods can throw a specific exception subclassed from Exception or RuntimeException

2.9. File Management

Through vulnerable file upload functionality an attacker can upload malicious files (Viruses, Worm, Trojan Horse and so on) to the server to disrupt the functioning of the server. An attacker can also perform DoS (Denial of Service) attack on the Web application by uploading large files (beyond the permissible upload size limit) causing the Web/Application server to crash.

- 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. Use white list approach for this.
- Validate uploaded files are the expected type by checking file headers. Checking for file type by extension alone is not sufficient

- Do not allow the users to upload a file in a Web space. Keep the Web space and the upload directory separate. It is advisable not to allow the files to get uploaded in the server directory and to ensure the files are uploaded outside the server directory.
- Limit the type of files that can be uploaded to only those types that are needed for business purposes
- Never allow the user to change the path of the location where the files are uploaded. Always take the path from the property file. Also, never expose the complete path location to the user, instead use the relative path.
- Turn off execution privileges on file upload directories
- Scan user uploaded files for viruses and malware

2.10. Resource Management and Synchronization

Computer resource includes CPU, memory, disk space, bandwidth, files, registry, process, socket, thread, sessions, cookies, data structure, communication pipes, and so on. Adversary can take advantage of race condition (for example, same resource is used by multiple process/threat for write operation), improper initialization (for example, reused sessions) or resource release (for example, data into temporary files) in program to exploit critical operations to corrupt data, launch denial of service or remote code execution attack or access sensitive data on file system.

- During declaration, initialise all variable with accepted default values. Do not perform initialisation for critical variables in try/catch block or exception handler.
- If variables are initialised directly from input received through external source, ensure that input is validated for known good values.
- Avoid performing complex calculations during initialisation. Use a language that enforces explicit initialisation of all variables before use.
- Check buffer boundaries if calling the function in a loop and make sure there is no danger of writing past the allocated space.
- Release all resources when they are no longer required. Memory should be allocated/freed using matching functions. Before releasing resource, check if resource exists. Use technologies that perform automatic garbage collection.
- While releasing complex object, ensure all member components are also disposed. Ensure that all pointers are set to NULL once their memory they point to has been freed.
- For multithreading applications, use synchronisation primitives. Only wrap these around critical-code to minimise the impact on performance. Use thread-safe functions on shared variables.
- Avoid compiler optimisation for threaded code. If this is not possible, use the volatile type modifier for critical variables to avoid unexpected compiler optimisation or reordering.

2.11. Program State Management

Programs that run as scheduled job, multi-step process or batch program has dependency on state data / inter process communication / external system response / service response to take execution forward. If application state or end user session data are stored in external access-controlled area (such as configuration files, profiles, cookies, hidden form fields, environment variables or registry keys) or locations controlled by end user, attacker can modify application state, change application's normal execution flow or retrieved sensitive data / restricted files or terminates entire program abruptly.

- Ensure application check for own state and user state data before processing and application must have rules for legitimate state transitions.
- Ensure state information is stored on server side only. If there is a need to store it on client, encrypt the data using Hash Message Authentication Code (HMAC) and put mechanism on the server side to detect tampering.
- For multi-step execution, ensure state change happens as per business rules and if failure happens in any step then program either wait for correction or recovery, or returns back to first step, or terminates gracefully.
- Execute code with least privilege into sandbox environment to enforce strict boundaries between the process and the operating system and restrict all access to files within a program directory. Validate authorisation before serving any request or executing any function.
- Reconcile data to ensure transactional integrity after batch or database transaction processing.

2.12. System Configuration

Configuration and related environment (especially for web applications) plays a key role in security. Configurations may include default credentials, user and group profiles, default file names, administrative information, protocol, database connection strings, logging details, dependency details, component configurations, external interface information, and default behaviour. Often, a wide gap between those who write the program and those who manage the environment leads to a variety of security problems. The installation, configuration, upgrade and changes shall be executed by authorized and competent administrators only.

- Remove test code or any functionality not intended for production, prior to deployment
- Prevent disclosure of your directory structure in the robots.txt file by placing directories not intended for public indexing into an isolated parent directory. Then "Disallow" that entire parent directory in the robots.txt file rather than Disallowing each individual directory
- Define which HTTP methods Get or Post, the application will support and whether it will be handled differently.
- Remove unnecessary information from HTTP response headers related to the OS, web-server version and application frameworks
- Do not use/release the product with test or default accounts; rather, configure the accounts
- During installation do not hardcode any backdoor accounts or special access mechanisms.
- Implement a software change control system to manage and record changes to the code both in development and production
- Secure the administrative interface by enforcing strong authentication (for example, by using certificates or IP-based restriction).
- Ensure that the programme and all dependent components execute with least privileges. Secure access for configuration files through operating system-level privilege. Change in configuration must follow the Change Management Process.