**Privacy and Data Security**

Privacy and data security safeguards help build end-user trust towards the system. Data security is the means of ensuring that data is kept safe from corruption, unauthorised 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. The degree of sensitivity accorded to personal and business data is relative and the level protection that a particular user considers adequate may prove to be excessive or unnecessary to some other user. Privacy and data security relies on data encryption/masking/tokenisation, legitimate disclosures, access control list, and data access permissions assigned on the storage systems (that includes file systems, backups and databases).

1. Mandate a design review if the software has privacy / legal / regulatory compliance requirements. Review data dictionary and identify private and sensitive data (even the DBA is not allowed to see it). Build a strong authentication and access control model for data storage and secret access. Employ permissions to restrict access (read and write) to data, secret and/or private information. Segregate the functions that deal with sensitive data and define special permissions for access to such data (for example, use SQL grant and revoke statements to restrict data access only to authorised users).
2. Avoid using home-grown cryptographic algorithms for data encryption; rather, use publicly scrutinised cryptographic algorithms (like AES) with recommended key sizes. Use OS specific mechanism (that is, DPAPI for windows and KeyChain for MAC) for key management.
3. Store only the information that is absolutely necessary (for example, ask users to enter their credit card number each time they perform a transaction, rather than encrypting and storing it). Ensure that encrypted data stored on a disk is not easy to decrypt (for example, database encryption is worthless if the database connection pool provides unencrypted access). Do not write sensitive data into an area that is accessible to users with low privileges.
4. Ensure client side caching is disabled for sensitive information (including autocomplete feature in web application forms). If temporary copy is created then ensure it is protected from unauthorised access.
5. 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. Keep such files containing secrets or passwords separate from application data. Wherever possible, use features like Key Vaults and periodically rotate the secrets. Use secure communication channels (such as SSL/TLS) to secure data in transit.
6. Provision method to remove sensitive data from the system at the end of required retention period.
7. Carefully design permissions for all install time and runtime objects (like file, configurations, stored procedure) and review the default access privileges for anonymous accounts and when privilege account is created.
8. 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.
9. Periodically monitor, validate and/or reconcile sensitive data items.
10. If data to be shared with third parties or consumed by external application/service, ensure that end user consent is taken prior to sharing data.
11. Maintain data access audit trails for personal/sensitive data to impose and periodically review it.
12. Ensure temporary files created /data acquired during processing / testing are always protected from unauthorized access / accidental deletion. Ensure they are removed as soon as process / test completes or within predefined period.

## Authorisation

Authorisation is the process of controlling access and rights to resources such as file, data, configuration, backups, program, and service for authenticated identities. Generally, authorisation is carried out after authentication and is imposed on the basis of policies or rules. Traditionally, authorisation is implemented through mandatory access control (MAC), discretionary access control (DAC), or role-based access control (RBAC). Improper or weak authorisation leads to information disclosure, abuse of functionality, and data tampering.

1. Create Role -> Functionality -> Data metrics during the design phase and implement strong authentication and access control measures around it. Further, maintain an audit trail (inclusive of the IP address) for sensitive operations.
2. Generate sensitive content on the fly and transmit it directly to the browser rather than saving it in the web server's file system. If the storage on the web server's file system is unavoidable, create temporary files with random filenames and delete them once they are no longer required.
3. Validate each request for authorisation; for web applications, ensure authorisation is checked before each page access with mechanism such as server side session token.
4. 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.
5. Restrict a normal user's access to system resources.

## 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 multi-factor authentication.

1. 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.
2. 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.
3. 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.
4. Protect the authentication token by using encryption and secure communication channels. Further, ensure that the token expires after a pre-determined period of time.
5. 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.
6. Use multi-factor authentication for critical applications.
7. Perform reconciliation activity periodically based on business requirements for accounts such as user accounts, service accounts, system accounts, bots and so on. Disable inactive accounts (i.e., accounts which are not used from certain period e.g., last 3 months).

## Audit Trail and Logging

1. 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 (such as Sarbanes-Oxley), intrusion detection, incident investigation, software trouble shooting, and program debugging. An audit trail is generally maintained using a database or in a log file.
2. Ensure that server time is synchronised with Network Time Protocol (NTP), other applications and systems. Ensure that all the records in the audit trail contain at least the timestamps, IP addresses, action owners, events and descriptions.
3. Prepare a list of sensitive events (such as authentication, authorisation, configuration changes, policy changes, and critical inputs) and ensure that they are properly logged. Further, all administrative operations and abnormal behaviour must be logged. Encrypt sensitive data before logging and avoid logging private data.
4. Use standard logging libraries (such as SLF4J, Log4J, Log4Net and so on) through wrapper and avoid custom or homegrown mechanisms for logging. Ensure that log files are stored in a safe location of the file system and that adequate disk space is available for logging. Log files to be set to Write Once and should not have execute permission. Use latest future proof encryption to encrypt the log files. Select an appropriate logging strategy such as date-wise logging (mostly-used), user-wise logging, or event-wise 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. If logs are being written in database, choose a different schema and apply encryption. Use a database user not having READ/UPDATE/DELETE/DROP access.
5. Ensure a provision to differentiate events from errors. Design the log format in such a way that the log can be easily imported to a database for review and analysis. Ensure that all events that include untrusted data will not execute as code in the intended log viewing software. It is recommended to make a copy of log before starting review.
6. Adhere to the applicable legal and regulatory standards requirements to maintain logs and their Hashes. Backup logs and analyse logs regularly.

**Using 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, risk, flexibility, cost, and so on. Cryptographic hardware and software must comply with applicable government regulations and geographical laws and they shall be adequately protected from unauthorised access or destruction.

1. Avoid using home-grown cryptographic algorithms; rather, only use publicly scrutinised cryptographic algorithms with recommended key sizes. Use future-proof algorithms. Please refer to the table 1 below for details.
2. Depending on business requirements, select the key length for encryption (for example, for mission critical systems, select a key length of 2048 bits or above). Use different keys for different applications and mandate activation for each key for limited period.
3. Use strong cryptographic hash algorithm (HMAC) to protect authenticity and integrity of message.
4. Do not hardcode keys into source code. Use secure storage (like 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.
5. Only use cryptographically strong PRNG to generate keys and session identifiers. Platform-specific APIs include: on Windows, use CryptoAPI; on UNIX, use /dev/urandom; and for java, use SecureRandom.
6. If the cryptographic library requires a temporary file, create file with a randomly-generated name in a secure area (for example, create file in user store instead of public store) and remove it after completing the task. To prevent race conditions while creating temporary files, use atomic creation functions (like mktemp() function/ command on Unix).
7. Ensure the server's certificate has a valid future date, 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). If public keys/certificates to be used, validate authenticity with issuer.
8. Use standard protocols to establish trust (such as SSL/TLS and Kerberos) while exchanging keys. Please refer to the table 2 below for details.
9. Configure server side certificate to support TLSv1 or above and do not support weak versions such as SSLv2.
10. Ensure that private information and secrets (like keys, certificates, and passwords) are stored securely. A secret can be split and stored at multiple locations such as configuration files or external servers and assembled at runtime. Data might be lost if encryption keys are lost or if disgruntled employees refuse to provide password. Therefore, ensure that the secret storage process is secure (this may involve some additional cost).
11. For proper key management in large infrastructure, deploy key manager for key generation, activation, storage, distribution, renewal, deactivation, recovery, deletion and define interface so that each application can connect with key manager.
12. Whenever there is a need to encrypt huge data, encrypt data with symmetric key, encrypt symmetric key with public key (asymmetric key) of recipient and send encrypted key along with encrypted data to recipient. (This will help improve performance).

| **Algorithm** | **Is Recommended?** | **Is Future-Proof?** | **Key Size (bits)** |
| --- | --- | --- | --- |
| AES | Yes | Yes | 128 minimum, 256+ advised |
| DES, 3DES | No | No | N/A |
| RSA | Yes | Yes | 1024 minimum, 2048+ advised |
| DSA | Yes | Yes | 1024 minimum, 2048+ advised |
| MD5 | No | No | N/A |
| SHA-1 | Yes | No | Only if mandated for compatibility and SHA-2 and above are not usable |
| SHA-2 | Yes | Yes | SHA-256 and above |

Table 2: Recommended Protocols

| **Protocol** | **Is Recommended?** | **Is Future-Proof?** |
| --- | --- | --- |
| SSLv2 | No | No |
| TLS | Yes | Yes |
| SSHv1 | No | No |
| SSHv2 | Yes | Yes |
| Kerberosv4 | Yes | No |
| Kerberosv5 | Yes | Yes |

## Input and Output Validation

Input validation is the process of validating the input received from user, subsystem component, or an external entity (like web service). Input validation is a challenging task, because the definition of what an appropriate input is changes with each software. However, a principle input received from the client should never be trusted and should be checked for value, integrity, and business rules. Input validation is the strongest countermeasure to prevent attacks like XSS, SQL injection, command injection, and other input attacks.

Output generated by the program is used either for display or as input to subsystem component, or an external entity. When the program generates an output in the form of a structured message containing result list, command, query, request or content, it needs to separately send control information and metadata from actual data. Most of the current systems bundle control data and actual data in the same stream. Adversary can send specially crafted input data to trick server to interpret input data as control information or metadata and generate unexpected output to benefit adversary.

1. Prepare a list of inputs that the software will process, such as user input, text file, feed, BLOB, and so on, and list the types of input that are not validated (with rationale). Identify all the potential interfaces through which software will received 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. Combine constrain (accept known good values as white list) and reject (reject known bad values as black list) strategy for validating or sanitising input.
2. Ensure all input to SQL interpreters, use parameterized interfaces, prepared statements, or are escaped properly.
3. Validate output for accuracy, completeness and required format. If output is being written to files then check for permission before writing/modifying files. Use appropriate validation mechanism (such as XSD for XML) to validate output.
4. 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.
5. 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. For web applications, use GET request only for navigation purposes; for everything else, use the POST request.
6. Use strongly typed parameterised query APIs with placeholder substitution markers even while calling stored procedures.
7. 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.
8. Ensure external inputs do not control application state, avoid using external input for file path creation and if that cannot be avoided, validate such inputs against known good values.

## Error and Exception Handling

Error and exception handling help 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 termination. Error/exception handling takes two forms: structured exception handling and functional error checking. During erroneous conditions, an exception generates a forced "subroutine call" from the statement that caused the error and transmits the call to a handler established by the program, runtime environment, or system. An unhandled error/exception discloses system and database error messages, stack trace, memory dump, and may even result in the program terminating abruptly. Proper error/exception handling can help prevent unintentional information leakage and attacks like enumeration, buffer overflow, and denial of service.

1. 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.
2. Disclose need to know information during error however ensure that information is adequate enough for end user to take action.
3. Production code should not be capable of producing debug messages or comments for developer. In the event of a failure, do not expose information (like verbose messages, stack traces, or path details) that could lead to information disclosure. If possible, configure the server with default error handler that returns an appropriately sanitised error message for all error paths.
4. If the language or framework supports a structured exception handler (such as try {} catch {}), it should be used in preference to functional error handling. Deploy centralised exception management strategy and ensure that all developers share a common approach to exception handling.
5. 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.
6. Use compilers that detect unhandled exceptions and return warning values at the time of compilation (for example, Visual Studio 2005 helps detect such issues).

## Configuration

Program configuration and related environment (especially for web applications) plays a key role in security. Program configurations may include authentication and authorisation details, default credentials, encryption details, user and group profiles, template locations, 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 authorised and competent administrators only.

1. Ensure that the default configurations (or switches) for every feature is secure and normal functionality does not rely on optionally installed features. Do not use default certificates.
2. 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.
3. Use secure communication channels such as SSL/TLS. Utilise SAML to encrypt and sign identity tokens in transit.
4. 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. Ensure that all security-relevant configuration information is stored in locations that are protected from unauthorized access and application access is denied if security configurations are not accessible to application.
5. Secure the administrative interface by enforcing strong authentication (for example, by using certificates or IP-based restriction). Maintain readonly audit trail for all configuration related changes. Deploy all infrastructure security measures.
6. Ensure configurations can be exported to human readable formats. Periodically monitor and validate all configurations.

## Privilege Management

Many a times, software (or component) requires performing operation with a privilege that is higher than the normal. If software runs as root or administrator, then all the normal security checks being performed by the operating system or surrounding environment are disabled. Adversary can abuse extra privileges to access restricted information, data store or configuration file. Vulnerability in privilege management can allow adversary to execute arbitrary command also.

1. Identify need for trust zones in the software and permissions for critical resources. Determine minimum privilege level required for execution of each software component and operation. Identify different permission for each resource such as read/write permission for file, socket creation, data access and provision functionality to maintain finer-grained control. Ensure that each component execute with designated minimal privileges.
2. If any component requires elevated privilege, ensure all such instances are known and documented. When operation with elevated privilege, raise privileges as late as possible and immediately drop once operation is done and verify that privileges are dropped properly.
3. Set most restrictive permissions for software during installations. During application initialisation, explicitly set the default permissions or unmask to a restricted setting so that application does not run with extra permission for privileged users.
4. Extensively perform validation if user input is to be used for privilege management functions or operating system APIs.
5. Log all privilege management events like assignment and alteration for user or process. Log all attempts for unauthorised access.

## Resource Management and Synchronisation

Computer resource includes CPU, memory, disk space, bandwidth, files, registry, process, socket, thread, handle, 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 initialisation (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.

1. 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 from 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.
2. Release all resources when they are no longer required. Memory should be allocated/freed using matching functions such as malloc/free, new/delete, and new[]/delete[]. Before releasing resource check if resource exists. Use technologies that perform automatic garbage collection.
3. While releasing complex object, ensure all member components are also disposed. Ensure that all pointers are set to NULL once they memory they point to has been freed.
4. 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. Do not allow interrupt or signal over critical parts of the code and but also make sure that the code does not go into a large or infinite loop.
5. 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.

## Program State Management

Programs that run as scheduled job, multi step process or wizard 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 file paths are constructed using invalidated user inputs (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.

1. Ensure integrity of program information configurations, environment settings, initialization data, license, dependency, certificate, system clock, location of update server and so on.
2. Ensure application check for own state and user state data before processing and application must has rules for legitimate state transitions.
3. When system generated data is used as input for internal processing, validate data before use.
4. 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.
5. For multi step execution, ensure state change happens as per business rules (Business rules should cater to sequence of execution, completeness, correctness, error and exception handling, etc) and if failure happens in any step then program either wait for correction or recovery, or returns back to first step, or terminates gracefully.
6. Execute code with list 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 authorization before serving any request or executing any function.
7. Where required, ensure that program reconciles data after processing. For example, reconcile data to ensure transactional integrity after batch or database transaction processing.

## Session Management

For multi user systems, users operate independently and it is important to maintain their state securely. States can be maintained in database, memory, file, and registry or within client. Stateless protocol like 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 (for example, variables). Often, the SID is not only used as an identification token, but also as an authenticator, this means that for the first request, users are authenticated with respect to their credentials; however, for all subsequent requests, the SID is used as an authenticator. The SID is communicated to the user through a request data, URL argument, a hidden form field, or a cookie.

1. 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.
2. 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.
3. Establish an appropriate session timeout period, periodically renew the SID, and renew the SID after a pre-determined number of session requests.
4. 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.
5. The software must be equipped with a log out feature. When a user logs out, invalidate the session and do not reuse the same SID for another user (this will cause performance penalty), and remove the SID from server. Depending on business requirements, ensure a reasonable duration for session timeout and prompt the user to re-authenticate, after a specified period of time.
6. Software must display time remaining to reauthenticate or notify user about session termination.