National Digital Health Mission (NDHM)

Secure Application Development – Reference Document

January 2021

1 Purpose

The purpose of this document is to provide NDHM's ecosystem partners with a secure coding guideline that may be referred to ensure security by design from the development phase. It is a technology agnostic set of general software security coding practices, that may be integrated into the development lifecycle. It is designed to serve as an easy reference, to help development teams quickly understand secure coding practices.

These guidelines are in place to prevent the introduction of security vulnerabilities, such as bugs and logic flaws in the application code or any programming language. This will further establish secure coding practices across NDHM ecosystem mitigating the risks that includes virus attacks, compromise or network systems, applications and services, and legal issues.

2 Application Secure Coding Guidelines

2.1 Web Application Security

2.1.1 Input Validation

1. Conduct all data validation on a trusted system (e.g., The server)
2. Identify all data sources and classify them into trusted and untrusted. Validate all data from untrusted sources (e.g., Databases, file streams, etc.)
3. There should be a centralized input validation routine for the application Specify proper character sets, such as UTF-8, for all sources of input
4. Encode data to a common character set before validating (Canonicalize) All validation failures should result in input rejection
5. Determine if the system supports UTF-8 extended character sets and if so, validate after UTF-8 decoding is completed
6. Validate all client provided data before processing, including all parameters, URLs and HTTP header content (e.g. Cookie names and values). Be sure to include automated post backs from JavaScript, Flash or other embedded code
7. Verify that header values in both requests and responses contain only ASCII characters \
8. Validate data from redirects (An attacker may submit malicious content directly to the target of the redirect, thus circumventing application logic and any validation performed before the redirect)
9. Validate for expected data types
10. Validate data range
11. Validate data length
12. Validate all input against a "white" list of allowed characters, whenever possible
13. If any potentially hazardous characters must be allowed as input, be sure that you implement additional controls like output encoding, secure task specific APIs and accounting for the utilization of that data throughout the application. Examples of common hazardous characters include: < > " ' % () & + \ \' \"

14. If your standard validation routine cannot address the following inputs, then they should be checked discretely

    a) Check for null bytes (%00)
    b) Check for new line characters (%0d, %0a, \r, \n)
    c) Check for “dot-dot-slash" (../ or ..\) , path alterations characters. In cases where UTF-8 extended character set encoding is supported, address alternate representation like: %c0%ae%c0%ae/ (Utilize canonicalization to address double encoding or other forms of obfuscation attacks).

2.1.2 Output Encoding

1. Conduct all encoding on a trusted system (e.g., The server)
2. Utilize a standard, tested routine for each type of outbound encoding
3. Contextually output encode all data returned to the client that originated outside the application's trust boundary. HTML entity encoding is one example, but does not work in all cases
4. Encode all characters unless they are known to be safe for the intended interpreter
5. Contextually sanitize all output of un-trusted data to queries for SQL, XML, and LDAP
6. Sanitize all output of un-trusted data to operating system commands

2.1.3 Authentication and Password Management

1. Require authentication for all pages and resources, except those specifically intended to be public
2. All authentication controls must be enforced on a trusted system (e.g., The server)
3. Establish and utilize standard, tested, authentication services whenever possible
4. Use a centralized implementation for all authentication controls, including libraries that call external authentication services
5. Segregate authentication logic from the resource being requested and use redirection to and from the centralized authentication control
6. All authentication controls should fail securely
7. All administrative and account management functions must be at least as secure as the primary authentication mechanism
8. If your application manages a credential store, it should ensure that only cryptographically strong one-way salted hashes of passwords are stored and that the table/file that stores the passwords and keys is write able only by the application. (Do not use the MD5 algorithm if it can be avoided)
9. Password hashing must be implemented on a trusted system (e.g., The server).
10. Validate the authentication data only on completion of all data input, especially for sequential authentication implementations
11. 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

12. Utilize authentication for connections to external systems that involve sensitive information or functions
13. Authentication credentials for accessing services external to the application should be encrypted and stored in a protected location on a trusted system (e.g., The server). The source code is NOT a secure location
14. Use only HTTP POST requests to transmit authentication credentials
15. Only send non-temporary passwords over an encrypted connection or as encrypted data, such as in an encrypted email. Temporary passwords associated with email resets may be an exception
16. Enforce password complexity requirements established by policy or regulation. Authentication credentials should be enough to withstand attacks that are typical of the threats in the deployed environment. (e.g., requiring the use of alphabetic as well as numeric and/or special characters)
17. Enforce password length requirements established by policy or regulation. Eight characters is commonly used, but 16 is better or consider the use of multi-word pass phrases
18. Password entry should be obscured on the user's screen. (e.g., on web forms use the input type "password")
19. Enforce account disabling after an established number of invalid login attempts (e.g., five attempts is common). The account must be disabled for a period of time sufficient to discourage brute force guessing of credentials, but not so long as to allow for a denial-of-service attack to be performed
20. Password reset and changing operations require the same level of controls as account creation and authentication.
21. Password reset questions should support sufficiently random answers. (e.g., "favorite book" is a bad question because “The Bible” is a very common answer)
22. If using email-based resets, only send email to a pre-registered address with a temporary link/password.
23. Temporary passwords and links should have a short expiration time
24. Enforce the changing of temporary passwords on the next use
25. Notify users when a password reset occurs
26. Prevent password re-use
27. Passwords should be at least one day old before they can be changed, to prevent attacks on password re-use
28. Enforce password changes based on requirements established in policy or regulation. Critical systems may require more frequent changes. The time between resets must be administratively controlled
29. Disable "remember me" functionality for password fields
30. The last use (successful or unsuccessful) of a user account should be reported to the user at their next successful login
31. Implement monitoring to identify attacks against multiple user accounts, utilizing the same password. This attack pattern is used to bypass standard lockouts, when user IDs can be harvested or guessed
32. Change all vendor-supplied default passwords and user IDs or disable the associated accounts
33. Re-authenticate users prior to performing critical operations
34. Use Multi-Factor Authentication for highly sensitive or high value transactional accounts
35. If using third party code for authentication, inspect the code carefully to ensure it is not affected by any malicious code

2.1.4 **Session Management**

1. Use the server or framework's session management controls. The application should only recognize these session identifiers as valid
2. Session identifier creation must always be done on a trusted system (e.g., The server)
3. Session management controls should use well vetted algorithms that ensure sufficiently random session identifiers
4. Set the domain and path for cookies containing authenticated session identifiers to an appropriately restricted value for the site
5. Logout functionality should fully terminate the associated session or connection
6. Logout functionality should be available from all pages protected by authorization
7. Establish a session inactivity timeout that is as short as possible, based on balancing risk and business functional requirements. Idle session timeout should be set at 15 minutes
8. Disallow persistent logins and enforce periodic session terminations, even when the session is active. Especially for applications supporting rich network connections or connecting to critical systems. Termination times should support business requirements and the user should receive sufficient notification to mitigate negative impacts
9. If a session was established before login, close that session and establish a new session after a successful login
10. Generate a new session identifier on any re-authentication
11. Do not allow concurrent logins with the same user ID
12. Do not expose session identifiers in URLs, error messages or logs. Session identifiers should only be in the HTTP cookie header. For example, do not pass session identifiers as GET parameters
13. Protect server-side session data from unauthorized access, by other users of the server, by implementing appropriate access controls on the server
14. Generate a new session identifier and deactivate the old one periodically. (This can mitigate certain session hijacking scenarios where the original identifier was compromised)
15. Generate a new session identifier if the connection security changes from HTTP to HTTPS, as can occur during authentication. Within an application, it is recommended to consistently utilize HTTPS rather than switching between HTTP to HTTPS.
16. Supplement standard session management for sensitive server-side operations, like account management, by utilizing per-session strong random tokens or parameters. This method can be used to prevent Cross Site Request Forgery attacks
17. Supplement standard session management for highly sensitive or critical operations by utilizing per-request, as opposed to per-session, strong random tokens or parameters
18. Set the "secure" attribute for cookies transmitted over an TLS connection
19. Set cookies with the HTTP Only attribute, unless you specifically require client-side scripts within your application to read or set a cookie's value

2.1.5 **Access Control**

1. Use only trusted system objects, e.g. server-side session objects, for making access authorization decisions
2. Use a single site-wide component to check access authorization. This includes libraries that call external authorization services
3. Access controls should fail securely
4. Deny all access if the application cannot access its security configuration information
5. Enforce authorization controls on every request, including those made by server-side scripts, "includes" and requests from rich client-side technologies like AJAX and Flash
6. Segregate privileged logic from other application code
7. Restrict access to files or other resources, including those outside the application's direct control, to only authorized users
8. Restrict access to protected URLs to only authorized users
9. Restrict access to protected functions to only authorized users
10. Restrict direct object references to only authorized users
11. Restrict access to services to only authorized users
12. Restrict access to application data to only authorized users
13. Restrict access to user and data attributes and policy information used by access controls
14. Restrict access security-relevant configuration information to only authorized users
15. Server-side implementation and presentation layer representations of access control rules must match
16. If state data must be stored on the client, use encryption and integrity checking on the server side to catch state tampering.
17. Enforce application logic flows to comply with business rules
18. Limit the number of transactions a single user or device can perform in a given period of time. The transactions/time should be above the actual business requirement, but low enough to deter automated attacks
19. Use the "referrer" header as a supplemental check only, it should never be the sole authorization check, as it is can be spoofed
20. If long authenticated sessions are allowed, periodically re-validate a user's authorization to ensure that their privileges have not changed and if they have, log the user out and force them to re-authenticate
21. 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.)
22. The application must support disabling of accounts and terminating sessions when authorization ceases (e.g., Changes to role, employment status, business process, etc.)
23. Service accounts or accounts supporting connections to or from external systems should have the least privilege possible
24. Create an Access Control Policy to document an application's business rules, data types and access authorization criteria and/or processes so that access can be properly provisioned and controlled. This includes identifying access requirements for both the data and system resources

2.1.6 **Cryptographic Practices**

1. All cryptographic functions used to protect secrets from the application user must be implemented on a trusted system (e.g., The server)

2. Protect master secrets from unauthorized access
3. Cryptographic modules should fail securely
4. 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
5. Cryptographic modules used by the application should be compliant to FIPS 140-2 or an equivalent standard. (See http://csrc.nist.gov/groups/STM/cmvp/validation.html)
6. Establish and utilize a policy and process for how cryptographic keys will be managed

2.1.7 Error Handling and Logging

1. Do not disclose sensitive information in error responses, including system details, session identifiers or account information
2. Use error handlers that do not display debugging or stack trace information
3. Implement generic error messages and use custom error pages
4. The application should handle application errors and not rely on the server configuration
5. Properly free allocated memory when error conditions occur
6. Error handling logic associated with security controls should deny access by default
7. All logging controls should be implemented on a trusted system (e.g., The server)
8. Logging controls should support both success and failure of specified security events
9. Ensure logs contain important log event data
10. Ensure log entries that include un-trusted data will not execute as code in the intended log viewing interface or software
11. Restrict access to logs to only authorized individuals
12. Utilize a master routine for all logging operations
13. Do not store sensitive information in logs, including unnecessary system details, session identifiers or passwords
14. Ensure that a mechanism exists to conduct log analysis
15. Log all input validation failures
16. Log all authentication attempts, especially failures
17. Log all access control failures
18. Log all apparent tampering events, including unexpected changes to state data
19. Log attempts to connect with invalid or expired session tokens
20. Log all system exceptions
21. Log all administrative functions, including changes to the security configuration settings
22. Log all backend TLS connection failures
23. Log cryptographic module failures
24. Use a cryptographic hash function to validate log entry integrity

2.1.8 Data Protection

1. Implement least privilege, restrict users to only the functionality, data and system information that is required to perform their tasks

2. Protect all cached or temporary copies of sensitive data stored on the server from unauthorized access and purge those temporary working files a soon as they are no longer required.
3. Encrypt highly sensitive stored information, like authentication verification data, even on the server side. Always use well vetted algorithms, see "Cryptographic Practices" for additional guidance
4. Protect server-side source-code from being downloaded by a user
5. Do not store passwords, connection strings or other sensitive information in clear text or in any non-cryptographically secure manner on the client side. This includes embedding in insecure formats like MS view state, Adobe flash or compiled code
6. Remove comments in user accessible production code that may reveal backend system or other sensitive information
7. Remove unnecessary application and system documentation as this can reveal useful information to attackers
8. Do not include sensitive information in HTTP GET request parameters
9. Disable auto complete features on forms expected to contain sensitive information, including authentication
10. Disable client-side caching on pages containing sensitive information. Cache-Control: no-store, may be used in conjunction with the HTTP header control "Pragma: no-cache", which is less effective, but is HTTP/1.0 backward compatible
11. The application should support the removal of sensitive data when that data is no longer required. (e.g. personal information or certain financial data)
12. Implement appropriate access controls for sensitive data stored on the server. This includes cached data, temporary files and data that should be accessible only by specific system users

2.1.9 **Communication Security**

1. Implement encryption for the transmission of all sensitive information. This should include TLS for protecting the connection and may be supplemented by discrete encryption of sensitive files or non-HTTP based connections
2. TLS certificates should be valid and have the correct domain name, not be expired, and be installed with intermediate certificates when required
3. Failed TLS connections should not fall back to an insecure connection
4. Utilize TLS connections for all content requiring authenticated access and for all other sensitive information
5. Utilize TLS for connections to external systems that involve sensitive information or functions
6. Utilize a single standard TLS implementation that is configured appropriately
7. Specify character encodings for all connections
8. Filter parameters containing sensitive information from the HTTP referrer, when linking to external sites

2.1.10 **System Configuration**

1. Ensure servers, frameworks and system components are running the latest approved version
2. Ensure servers, frameworks and system components have all patches issued for the version in use

3. Turn off directory listings
4. Restrict the web server, process and service accounts to the least privileges possible
5. When exceptions occur, fail securely
6. Remove all unnecessary functionality and files
7. Remove test code or any functionality not intended for production, prior to deployment
8. 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
9. Define which HTTP methods, Get or Post, the application will support and whether it will be handled differently in different pages in the application
10. Disable unnecessary HTTP methods, such as WebDAV extensions. If an extended HTTP method that supports file handling is required, utilize a well-vetted authentication mechanism
11. If the web server handles both HTTP 1.0 and 1.1, ensure that both are configured in a similar manor or insure that you understand any difference that may exist (e.g. handling of extended HTTP methods)
12. Remove unnecessary information from HTTP response headers related to the OS, web-server version and application frameworks
13. Implement an asset management system and register system components and software in it
14. Isolate development environments from the production network and provide access only to authorized development and test groups. Development environments are often configured less securely than production environments and attackers may use this difference to discover shared weaknesses or as an avenue for exploitation
15. Implement a software change control system to manage and record changes to the code both in development and production

2.1.11 **Database Security**

1. Use strongly typed parameterized queries
2. Utilize input validation and output encoding and be sure to address meta characters. If these fails, do not run the database command
3. Ensure that variables are strongly typed
4. The application should use the lowest possible level of privilege when accessing the database
5. Use secure credentials for database access
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.
7. Close the connection as soon as possible
8. Remove or change all default database administrative passwords. Utilize strong passwords/phrases or implement multi-factor authentication
9. 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 (surface area reduction))
10. Remove unnecessary default vendor content (e.g., sample schemas)
11. Disable any default accounts that are not required to support business requirements
12. The application should connect to the database with different credentials for every trust distinction (e.g., user, read-only user, guest, administrators)

2.1.12 **File Management**

1. Do not pass user supplied data directly to any dynamic include function
2. Require authentication before allowing a file to be uploaded
3. Limit the type of files that can be uploaded to only those types that are needed for business purposes
4. Validate uploaded files are the expected type by checking file headers. Checking for file type by extension alone is not sufficient
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.
6. Prevent or restrict the uploading of any file that may be interpreted by the web server.
7. Turn off execution privileges on file upload directories
8. Implement safe uploading in UNIX by mounting the targeted file directory as a logical drive using the associated path or the chrooted environment
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
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
11. Do not pass directory or file paths, use index values mapped to pre-defined list of paths
12. Never send the absolute file path to the client
13. Ensure application files and resources are read-only
14. Scan user uploaded files for viruses and malware

2.1.13 **Memory Management**

1. Utilize input and output control for un-trusted data
2. Double check that the buffer is as large as specified
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
4. Check buffer boundaries if calling the function in a loop and make sure there is no danger of writing past the allocated space
5. Truncate all input strings to a reasonable length before passing them to the copy and concatenation functions
6. Specifically, close resources, don’t rely on garbage collection. (e.g., connection objects, file handles, etc.)
7. Use non-executable stacks when available
8. Avoid the use of known vulnerable functions (e.g., printf, strcat, strcpy etc.)
9. Properly free allocated memory upon the completion of functions and at all exit points

2.2 Mobile Application Security

1. Verify all application components are identified and are known to be needed.
2. Security controls should never be enforced only on the client side, but on the respective remote endpoints.
3. A high-level architecture for the mobile app and all connected remote services should be defined and security should be addressed in that architecture.
4. Ensure that the data considered sensitive in the context of the mobile app is clearly identified.
5. All app components should be defined in terms of the business functions and/or security functions they provide.
6. A threat model for the mobile app and the associated remote services, which identifies potential threats and countermeasures, should be produced.
7. All security controls should have a centralized implementation.
8. An explicit policy for how cryptographic keys (if any) are managed, and the lifecycle of cryptographic keys is enforced should be present. Ideally, follow a key management standard such as NIST SP 800-57.
9. A mechanism for enforcing updates of the mobile app should exist.
10. Security should be addressed within all parts of the software development lifecycle.
11. Verify that system credential storage facilities are used appropriately to store sensitive data, such as PII, user credentials or cryptographic keys.
12. No sensitive data should be stored outside of the app container or system credential storage facilities.
13. Ensure that no sensitive data is written to application logs.
14. Ensure that no sensitive data is shared with third parties unless it is a necessary part of the architecture.
15. Ensure that the keyboard cache is disabled on text inputs that process sensitive data.
16. Ensure that no sensitive data is exposed via IPC mechanisms.
17. Ensure that no sensitive data, such as passwords or pins, is exposed through the user interface.
18. Ensure that no sensitive data is included in backups generated by the mobile operating system.
19. Ensure that the app removes sensitive data from views when backgrounded.
20. The app should not hold sensitive data in memory longer than necessary, and memory is cleared explicitly after use.
21. Ensure that the app enforces a minimum device-access-security policy, such as requiring the user to set a device passcode.
22. The app should educate the user about the types of personally identifiable information processed, as well as security best practices the user should follow in using the app.
23. The app should not rely on symmetric cryptography with hardcoded keys as a sole method of encryption.
24. Ensure that the app uses proven implementations of cryptographic primitives.
25. Ensure that the app uses cryptographic primitives that are appropriate for the use-case, configured with parameters that adhere to industry best practices.
26. Ensure that the app does not use cryptographic protocols or algorithms that are widely considered depreciated for security purposes.
27. Ensure that the app doesn't re-use the same cryptographic key for multiple purposes.
28. Ensure that all random values are generated using a sufficiently secure random number generator.
29. If the app provides users with access to a remote service, an acceptable form of authentication such as username/password authentication should be performed at the remote endpoint.

30. Ensure that the remote endpoint uses randomly generated session identifiers, if classical server-side session management is used, to authenticate client requests without sending the user's credentials.
31. Ensure that the remote endpoint uses server-side signed tokens, if stateless authentication is used, to authenticate client requests without sending the user's credentials.
32. Ensure that the remote endpoint terminates the existing session when the user logs out.
33. Ensure that a password policy exists and is enforced at the remote endpoint.
34. Ensure that the remote endpoint implements a mechanism to protect against the submission of credentials an excessive number of times.
35. Ensure that sessions are terminated at the remote endpoint or tokens expire after a predefined period of inactivity.
36. Verify that biometric authentication, if any, is not event-bound (i.e. using an API that simply returns "true" or "false"). Instead, it is based on unlocking the keychain/keystore.
37. Ensure that a second factor of authentication exists at the remote endpoint and the 2FA requirement is consistently enforced.
38. Ensure that step-up authentication is required to enable actions that deal with sensitive data or transactions.
39. Ensure that the app informs the user of all login activities with his or her account. Users are able view a list of devices used to access the account, and to block specific devices.
40. Ensure that data is encrypted on the network using TLS. The secure channel is used consistently throughout the app.
41. Ensure that the TLS settings are in line with current best practices, as far as they are supported by the mobile operating system.
42. Ensure that the app verifies the X.509 certificate of the remote endpoint when the secure channel is established. Only certificates signed by a valid CA are accepted.
43. Ensure that the app either uses its own certificate store, or pins the endpoint certificate or public key, and subsequently does not establish connections with endpoints that offer a different certificate or key, even if signed by a trusted CA.
44. Ensure that the app doesn't rely on a single insecure communication channel (email or SMS) for critical operations, such as enrollments and account recovery.
45. Ensure that the app only depends on up to date connectivity- and security libraries.
46. Ensure that the app only requires the minimum set of permissions necessary.
47. All inputs from external sources and the user should be validated and if necessary sanitized. This includes data received via the UI, IPC mechanisms such as intents, custom URLs, and network sources.
48. The app should not export sensitive functionality via custom URL schemes, unless these mechanisms are properly protected.
49. The app should not export sensitive functionality through IPC facilities, unless these mechanisms are properly protected.
50. Ensure that JavaScript is disabled in WebViews unless explicitly required.
51. Ensure that WebViews are configured to allow only the minimum set of protocol handlers required (ideally, only https). Potentially dangerous handlers, such as file, tel and app-id, are disabled.
52. If native methods of the app are exposed to a WebView, ensure that the WebView only renders JavaScript contained within the app package.
53. Ensure that object serialization, if any, is implemented using safe serialization APIs.
54. Code Quality and Build Settings

55. Ensure that the app is signed and provisioned with valid certificate.
56. Ensure that the app has been built in release mode, with settings appropriate for a release build (e.g. non-debuggable).
57. Ensure that debugging symbols have been removed from native binaries.
58. Ensure that debugging code has been removed, and the app does not log verbose errors or debugging messages.
59. Ensure that all third-party components used by the mobile app, such as libraries and frameworks, are identified, and checked for known vulnerabilities.
60. Ensure that the app catches and handles possible exceptions.
61. Ensure that error handling logic in security controls denies access by default.
62. Ensure that in unmanaged code, memory is allocated, freed and used securely.
63. Free security features offered by the toolchain, such as bytecode minification, stack protection, PIE support and automatic reference counting, are activated.
64. Implement two or more functionally independent methods of root detection and respond to the presence of a rooted device either by alerting the user or terminating the app.
65. Implement multiple functionally independent debugging defences that, in context of the overall protection scheme, force adversaries to invest significant manual effort to enable debugging. All available debugging protocols must be covered (e.g. JDWP and native).
66. Application should detect and respond to,
    i. tampering with executable files and critical data.
    ii. being run in an emulator using any method.
    iii. modifications of process memory, including relocation table patches and injected code.
67. Application should detect the presence of widely used reverse engineering tools, such as code injection tools, hooking frameworks and debugging servers.
68. Implement multiple different responses to tampering, debugging and emulation, including stealthy responses that don't simply terminate the app.
69. The detection mechanisms should trigger responses of different types, including delayed and stealthy responses.
70. Obfuscating transformations and functional defences should be interdependent and well-integrated throughout the app, in case of iOS app.
71. While focusing on Android app, obfuscation should be applied to programmatic defences, which in turn impedes de-obfuscation via dynamic analysis.
72. Implement a 'device binding' functionality when a mobile device is treated as being trusted. Verify that the device fingerprint is derived from multiple device properties.
73. All the executable files and libraries belonging to the app are either encrypted on the file level and/or important code and data segments inside the executables are encrypted or packed. Trivial static analysis does not reveal important code or data.
74. Verify that if the goal of obfuscation is to protect sensitive computations, an obfuscation scheme is used that is both appropriate for the task and robust against manual and automated de-obfuscation methods, considering currently published research. The effectiveness of the obfuscation scheme must be verified through manual testing. Note that hardware-based isolation features are preferred over obfuscation whenever possible.

2.3 Application Programming Interface (API) Security

1. Implement a proper authorization mechanism that relies on the user policies and hierarchy.

2. Prefer not to use an ID that has been sent from the client, but instead use an ID that is stored in the session object when accessing a database record by the record ID.
3. Use an authorization mechanism to check if the logged-in user has access to perform the requested action on the record in every function that uses an input from the client to access a record in the database.
4. Prefer to use random and unpredictable values as GUIDs for records' IDs.
5. Write tests to evaluate the authorization mechanism. Do not deploy vulnerable changes that break the tests.
6. Make sure that all the possible flows to authenticate to the API are known (mobile/ web/deep links that implement one-click authentication/etc.)
7. Don't reinvent the wheel in authentication, token generation, password storage. Use the standards.
8. Credential recovery / forget password endpoints should be treated as login endpoints in terms of brute force, rate limiting and lockout protections.
9. Where possible, implement multi-factor authentication.
10. Implement anti brute force mechanisms to mitigate credential stuffing, dictionary attack and brute force attacks on the authentication endpoints. This mechanism should be stricter than the regular rate limiting mechanism on your API.
11. Implement account lockout & captcha mechanism to prevent brute force against specific users.
12. Implement weak-password checks.
13. API keys should not be used for user authentication, but for client app / project authentication.
14. Never rely on the client side to perform sensitive data filtering.
15. Review the responses from the API to make sure they contain only legitimate data.
16. Explicitly define and enforce data returned by all API methods, including errors. Whenever possible use schemas for responses, patterns for all strings and clear field names.
17. Define all sensitive and personally identifiable information (PII) that your application stores and works with and review all API calls returning such information to see if these responses can be a security issue.
18. Docker makes it easy to limit memory, CPU, number of restarts, file descriptors, and processes.
19. Implement a limit on how often a client can call the API within a defined timeframe.
20. For sensitive operations such as login or password reset, consider rate limits by API method (e.g., authentication), client (e.g., IP address), property (e.g., username).
21. Notify the client when the limit is exceeded by providing the limit number and the time at which the limit will be reset.
22. Add proper server-side validation for query string and request body parameters, specifically the one that controls the number of records to be returned in the response.
23. Define and enforce maximum size of data on all incoming parameters and payloads such as maximum length for strings and maximum number of elements in arrays.
24. If your API accepts compressed files check compression ratios before expanding the files to protect yourself against "zip bombs".
25. Application should have a consistent and easy to analyse authorization module that is invoked from all the business functions. Frequently, such protection is provided by one or more components external to the application code.
26. The enforcement mechanism(s) should deny all access by default, requiring explicit grants to specific roles for access to every function.
27. Review your API endpoints against function level authorization flaws, while keeping in mind the business logic of the application and groups hierarchy.

28. Make sure that all your administrative controllers inherit from an administrative abstract controller that implements authorization checks based on the user's group/role.
29. Make sure that administrative functions inside a regular controller implements authorization checks based on the user's group and role.
30. If possible, avoid using functions that automatically bind a client's input into code variables or internal objects.
31. Whitelist only the properties that should be updated by the client.
32. Use built-in features to blacklist properties that should not be accessed by clients.
33. If applicable, explicitly define and enforce schemas for the input data payloads.
34. The API life cycle should include:
    i. A repeatable hardening process leading to fast and easy deployment of a properly locked down environment.
    ii. A task to review and update configurations across the entire API stack. The review should include orchestration files, API components, and cloud services (e.g., S3 bucket permissions).
    iii. A secure communication channel for all API interactions access to static assets (e.g., images).
    iv. An automated process to continuously assess the effectiveness of the configuration and settings in all environments.
    v. To prevent exception traces and other valuable information from being sent back to attackers, if applicable, define and enforce all API response payload schemas including error responses.

35. Perform data validation using a single, trustworthy, and actively maintained library.
36. Validate, filter, and sanitize all client-provided data, or other data coming from integrated systems.
37. Special characters should be escaped using the specific syntax for the target interpreter.
38. Prefer a safe API that provides a parameterized interface.
39. Always limit the number of returned records to prevent mass disclosure in case of injection.
40. Validate incoming data using sufficient filters to only allow valid values for each input parameter.
41. To prevent data leaks, define and enforce schemas for all API responses.
42. Define data types and strict patterns for all string parameters.
43. Inventory all API hosts and document important aspects of each one of them, focusing on the API environment (e.g., production, staging, test, development), who should have network access to the host (e.g., public, internal, partners) and the API version.
44. Inventory integrated services and document important aspects such as their role in the system, what data is exchanged (data flow), and its sensitivity.
45. Document all aspects of your API such as authentication, errors, redirects, rate limiting, cross-origin resource sharing (CORS) policy and endpoints, including their parameters, requests, and responses.
46. Generate documentation automatically by adopting open standards. Include the documentation build in your CI/CD pipeline.
47. Make API documentation available to those authorized to use the API.
48. Use external protection measures such as API security firewalls for all exposed versions of your APIs, not just for the current production version.

49. Avoid using production data with non-production API deployments. If this is unavoidable, these endpoints should get the same security treatment as the production ones.
50. When newer versions of APIs include security improvements, perform risk analysis to make the decision of the mitigation actions required for the older version: for example, whether it is possible to backport the improvements without breaking API compatibility or you need to take the older version out quickly and force all clients to move to the latest version.
51. Log all failed authentication attempts, denied access, and input validation errors.
52. Logs should be written using a format suited to be consumed by a log management solution and should include enough detail to identify the malicious actor.
53. Logs should be handled as sensitive data, and their integrity should be guaranteed at rest and transit.
54. Configure a monitoring system to continuously monitor the infrastructure, network, and the API functioning.
55. Use a Security Information and Event Management (SIEM) system to aggregate and manage logs from all components of the API stack and hosts.
56. Configure custom dashboards and alerts, enabling suspicious activities to be detected and responded to earlier.

2.4 General Coding Practices

1. A unique identifier (e.g., user ID, digital certificate) must be associated with each user of the NDHM applications. Information that is subject to data privacy regulations, such as Aadhaar number (UID) are not to be used as user IDs.
2. Use tested and approved managed code rather than creating new unmanaged code for common tasks
3. 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 using application-initiated command shells
4. Use checksums or hashes to verify the integrity of interpreted code, libraries, executables, and configuration files
5. Utilize locking to prevent multiple simultaneous requests or use a synchronization mechanism to prevent race conditions
6. Protect shared variables and resources from inappropriate concurrent access
7. Explicitly initialize all your variables and other data stores, either during declaration or just before the first usage
8. In cases where the application must run with elevated privileges, raise privileges as late as possible, and drop them as soon as possible
9. Do not pass user supplied data to any dynamic execution function
10. Restrict users from generating new code or altering existing code
11. Review all secondary applications, third party code and libraries to determine business necessity and validate safe functionality, as these can introduce new vulnerabilities
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
13. Input data streams should be sanitized to remove character types that are not intended for the input field (e.g., name field should only contain alphabetic characters, etc.).
14. The buffer size of an input field should be set to the maximum length of characters expected as input (e.g., users should not be allowed to enter more characters than the buffer size of the field).
15. Code should be free of trap doors and back doors which allow access to the application by circumventing or bypassing the access control system. (Note: this includes back doors designed into the application as well as tools external to the application).
16. Digital certificates used for identification (If any) must be issued by a Certificate Authority
17. NDHM confidential information related to non-public financial information, and personal information such as financial or health records, must be encrypted if sent electronically across the Internet.
18. Always enforce the Integrity of the sensitive code. Lock the code and once the code is locked, Integrity validations must be performed at all stages, to ensure the code is not modified in Test/production systems.
19. The application developers should add a mandatory privacy tick-box to ensure that the user was notified about the Terms and Conditions and Data Privacy Policy or notice of NDHM. It is highly recommended that the description line of the checkbox should contain a hyperlink that will redirect to the appropriate section on NDHM website.

3 Key Design Guidelines to be adhered

3.1.1 Requirement 1: Reusable passwords used in identity verification challenges must adhere to the following syntax rules:

a) Passwords shall not be hardcoded in codes, login scripts, any executable program or files or included in any automated log-on process, e.g. stored in a macro or function key
b) Password should not be stored or transmitted in application, over the Internet, public networks, or wireless devices in clear text or in any reversible form
c) Passwords should be encrypted, if possible, when stored in applications and data repositories. If passwords cannot be encrypted, access to the file or database element containing the passwords must be restricted to only authorized application security administrators
d) System should not allow the username and password to be the same for a particular user
e) Passwords should not be remembered in cache or browser memory
f) Password policy in the application shall be designed as per NHA External Ecosystem – IS Policy.

3.1.2 Requirement 2: Authentication Tokens (if any) used in identity verification challenges must be protected as follows:

a) The default token lifetime for application administrative users must not exceed 12 hours, and for general users, must not exceed 30 hours. If necessary, a locally administered process can be established to provide longer default token lifetimes.
b) All requests for services or information access must be based on either an authentication token or user ID and password. An authentication process which
   - is based on token issuance using the user ID and password, and
   - encrypted token exchanging for access to information is permitted.

3.1.3 Requirement 3: Audit records are to be created for all application and data repository.

The following specification must be included in the implementation of this requirement:

a) Both successful and unsuccessful logon access attempt activities are to be logged.
b) Activities performed using security administrative authority (e.g., adds/changes/deletes of user access rights, changes to security configurations, etc.) Are to be logged. Logging of these activities must never be turned off.
c) Audit records must include date, time, type of access attempt, IP address of client and user identifier.
d) Audit trail records must include identification of who made the change, the time and date of the change, the data content before the change, and the data content after the change.
e) Audit records must be retained for 3 months online and 2 years in archival form. Only in case of Aadhaar transaction logs are to be retained as per the following:
   - 2 (two) years - Online
   - 5 (five) years - Archived

- Upon expiration - Deleted (except those records required to be retained by a court or required to be retained for any pending disputes)

f) Audit trail records must be maintained for a period of time determined by respective business process owner.

g) In case of NDHM Application conducting Aadhaar Authentication and collecting Aadhaar E-KYC information of the beneficiaries/employees/partners, they must mandatorily store comprehensive logs with respect to Aadhaar transactions. Storage of Aadhaar logs has been mandated in NDHM IS policy. The following specification must be included in the implementation of this requirement:
   - The following parameters are to be mandatorily stored in Aadhaar transaction logs as captured in "Request sent to UIDAI":
     - 'tid' – use of registered device
     - 'ac' – aua code
     - 'sa' – sub-aua code
     - 'ver' – API version
     - 'txn' – transaction id
     - 'ts' – timestamp
     - Unique terminal ID of the device from which authentication was requested
     - Authentication Modality (OTP/OTP/Iris)
     - 'rd' - The record of disclosure of information to the Aadhaar number holder at the time of authentication
     - 'rc' - Record of consent of the Aadhaar number holder for authentication
   - The following parameters are to be mandatorily stored in Aadhaar transaction logs as captured in "Response received from UIDAI:
     - UID token
     - 'ts' – response timestamp
     - Authentication status
     - Error code
     - e-kyc information
   - The following parameters must not be stored in Aadhaar transactions log from the request sent to UIDAI:
     - Parameters of authentication request submitted - all fields sent to UIDAI except:
     - Aadhaar number
     - PID block capturing beneficiary biometric data
     - Session key (skey)
     - HMAC.
   - Aadhaar number must not be stored in Aadhaar transactions log from the response received from UIDAI:

3.1.4 Requirement 4: In case any application is conducting Aadhaar Authentication of the beneficiaries/employees/partners, then the application must mandatorily collect consent;

The following specifications must be clear in the Aadhaar consent statement:

a) The consent shall be designed as per the requirement of Aadhaar Act 2016 and its regulations
b) The nature of information that will be shared by the UIDAI upon authentication;
c) The uses to which the information received during Aadhaar authentication may be put
d) Alternatives to submission of identity information.
e) Consent shall be obtained in the language of the individual
f) Aadhaar number holder shall be informed about consent revocation

3.1.5 Requirement 5: Maintain an unique transaction number for every transaction in the application

3.1.6 Requirement 6: Ensure the application is developed in a manner to comply with NDHM HDM policy & External Ecosystem – IS Policy.

Annexure: Reference URLs

1. https://ndhm.gov.in/documents/HealthDataManagementPolicy
2. https://uidai.gov.in/media-resources/uidai-documents/circulars-memorandums-notification.html
3. https://uidai.gov.in/ecosystem/authentication-ecosystem/authentication-requesting-agency.html
4. OWASP Web Application Security Top 10
   https://owasp.org/www-project-top-ten/
5. OWASP Mobile Application Security Top 10 2019
   https://owasp.org/www-project-mobile-top-10/
6. OWASP API Security Top 10 2019
   https://owasp.org/www-project-api-security/

Document Control

Type of Information	Document Data
Title	Secure Coding and Application Development
Document version	1.0
File No	XXXX
Document Owner	National Health Authority (NDHM)

Document Approver

Version	Date of Approval	Approved By	Name
1.0	January 2021		

Revision Change History

Version	Date	Prepared By / Modified By	Significant Changes
1.0	January 2021	NHA IS team	Initial documentation