SECURE SOFTWARE DEVELOPMENT LIFE CYCLE

SOCIETY HUB

Vineet Rao - 161070001

Ameya Daddikar - 161070015

Aim

To implement a security framework (secure software lifecycle) for Society Management System (SocietyHub).

Phases	
	Description: Society Hub is a web platform for housing societies to delegate a number of management responsibilities to. It aids in easy storage and retrieval of information related to various parts of the society, along with an account system for added security.

Analysis	• Account system for each flat. • Access to profile details, and issues raised. • Viewing of latest notices pertaining to the society • Completely responsive web design, and can easily be scaled along with the database. • Privileges can be given to administrator and program manager only so that anyone else can not misuse those privileges. • Special accounts for Community Members and Society Admins that can access Society Admin Pages to manage the society	
Design	Input Validation	• Validate and sanitize all input data • Ensure validation on trusted system (server) • Prevent unsanitized data from being passed directly to query/command evaluation engines • Always use minimal user-input data
	Authentication and Password Management	• All access to a society's data and pages must be secured with authentication • All authentication must be performed on a trusted system • Ensure any failure is secure • Store passwords in a hashed manner, on a centralized system.
	Authorization	• Assign roles to users (member, building manager, society manager, committee member, etc.)
	Sensitive details management	• Sensitive identifiable data must be protected • Show only to authorized users • Try to store hashes were actual data is not needed

	Session management	● Create session identifier for every login, at trusted system ● Ensure finite time limit for session ● Invalidate session identifier on logout, regardless of timeout
	Cryptographic practices	● Use hashing for storage of sensitive data like passwords ● Prefer encrypted communication (HTTPS) ● Ensure that master secrets are stored securely (especially confiedential credentials) ● Use compliant cryptographic modules. ● Use only module functionality for cryptographic functions like random number generation, hashing, signature, etc.
	Exception management	● Exception should be handled without causing program failure ● No sensitive data should be displayed to the user ● Maintain logs of every exception
	Database Security	● Use prepared statements ● Ensure variables used as parameters are strongly type ● Use minimum connection time ● Where possible, use views and stored procedures as opposed to base tables
Implementation	Input Validation	1. Conduct all data validation on server side. 2. 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 utilisation of that data throughout the application . Examples of common hazardous characters include: < > " ' % ( ) & + \ \' \" 3. 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 4. Verify that header values in both requests and responses contain only ASCII characters 5. Utilize canonicalization to address double encoding or other forms of obfuscation attacks)

	Authentication	1. Require authentication for all pages and resources, except those specifically intended to be public 2. Establish and utilize standard, tested, authentication services whenever possible 3. Use a centralized implementation for all authentication controls, including libraries that call external authentication services 4. If your application manages a credential store, it should ensure that only cryptographically strong oneway 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) 5. Authentication failure responses should not indicate which part of the authentication data was incorrect 6. Use only HTTP POST requests to transmit authentication credentials 7. Enforce password complexity requirements established by policy or regulation. Authentication credentials should be sufficient 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) 8. Password reset and changing operations require the same level of controls as account creation and authentication. 9. If using email based resets, only send email to a pre-registered address with a temporary link/password 10. Notify users when a password reset occurs 11. 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

	Authorization	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. 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 4. Segregate privileged logic from other application code 5. Restrict access to files or other resources, including those outside the application's direct control, to only authorized users 6. Restrict access to protected URLs to only authorized users 7. Restrict access to protected functions to only authorized users 8. Restrict access to services to only authorized users 9. Restrict access to application data to only authorized users 10. Restrict access to user and data attributes and policy information used by access controls 11. If state data must be stored on the client, use encryption and integrity checking on the server side to catch state tampering. 12. Use the "referer" header as a supplemental check only, it should never be the sole authorization check, as it is can be spoofed 13. 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

	Sensitive management	1. Do not disclose sensitive information in error responses, including system details, session identifiers or account information 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 4. Do not include sensitive information in HTTP GET request parameters 5. Disable auto complete features on forms expected to contain sensitive information, including authentication 6. 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
	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. Logout functionality should fully terminate the associated session or connection 5. 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. 6. Protect server side session data from unauthorized access, by other users of the server, by implementing appropriate access controls on the server 7. Generate a new session identifier on any re-authentication 8. 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
	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)
	Exception management	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. Properly free allocated memory when error conditions occur 4. All logging controls should be implemented on a trusted system (e.g., The server) 5. Error handling logic associated with security controls should deny access by default
Testing	● Test based on test data and cases generated from UML diagrams. ● Perform vulnerability scanning for common ones, and known vulnerabilities in used libraries ● Test variety of input data, valid as well as invalid ● Make requests to non-existent pages, use wrong HTTP method, to check error messages ● Attempt unauthorized activities with a different authentication, like adding buildings from a normal member account.	

Deployment	Network Threats: All network guards like firewall, application firewall, honey-pot and IDS should be updated otherwise following threats are present Information gathering, Sniffing or eavesdropping , spoofing, Session hijacking, Denial of service Server Threats : Server on which I am going to deploy “SocietyHub” should be secure otherwise following threats are possible Viruses , Trojan horse and worms Foot printing Password cracking Denial of service Arbitrary code execution Unauthorized access. Common Procedures: 1. Remove test code or any functionality not intended for production, prior to deployment 2. Collaboration Between Development and Operations 3. Build & Release Automation 4. Minimize the Amount of Change 5. Create and Test SQL Change Scripts 6. Setup Synthetic Transactions Tests 7. Setup network guards like firewalls, application firewalls, IDS, etc.
Maintenance	● Raise alerts on repeated attempts to access unauthorized requests. ● All tables used in “SocietyHub” should be updated properly. ● Allow society Admins to raise issue with website to block some user if necessary. ● Regular system backups and checks