KPMG

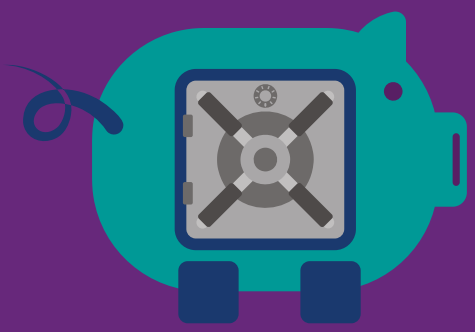
Secure Development Guidelines

KPMG (India)

Confidential

Version – 1.0

May 2017



Document Control

Document Name: Secure Development Guideline

Version history

Version	Author	Date	Detail of Change
1.0	Ruby Mishra	July 2016	First release
1.0	Shikha Chandravanshi	May 2017	Annual review

Version	Reviewed By	Date	Detail of Change
1.0	Ritam Guar	26 July 2016	First release
1.0	Sanjeev Garg	26 July 2016	First release
1.0	Ritam Gaur	01July 2017	First release

This document has been approved by

Version	Approved By	Date of review/approval
1.0	George Mathew	27 July 2016
1.0	Ruby Mishra	29 May 2017

KPMG

Glossary

Asset	Anything that has value to an organization
Control	Means of managing risk, including policies, procedures, guidelines, practices or organizational structures, which can be of administrative, technical, management or legal nature

KPMG

Guideline	A description that clarifies what should be done and how, to achieve the objectives set out in policies
Information Processing Facilities	Any information processing system, service or infrastructure, or the physical locations housing them
Information Security	Preservation of confidentiality, integrity and availability of information; in addition, other properties, such as authenticity, accountability, non – repudiation, and reliability can also be involved
Information Security Event	An information security event is an identified occurrence of a system, service or network state indicating a possible breach of information security policy or failure of safeguards, or a previously unknown situation that may be security relevant
Information Security Incident	An information security incident is indicated by a single or a series of unwanted or unexpected information security events that have a significant probability of compromising business operations and threatening information security

Policy	Overall intention and direction as formally expressed by management
Risk	Combination of probability of occurrence of an event and its consequence
Risk Analysis	Systematic use of information to identify sources and to estimate the risk
Risk Assessment	Overall process of risk analysis and risk evaluation
Risk Evaluation	Process of comparing the estimated risk against given risk criteria to determine the significance of the risk
Risk Management	Coordinated activities to direct and control an organization with regard to risk
Risk Treatment	Process of selection and implementation of measures to modify risk

Threat	A potential cause of an unwanted incident, which may result in harm to the organization
Vulnerability	A weakness of an asset or a group of assets that can be exploited by one or more threats
Third Party	That person or body that is recognized as being independent of the parties involved, as concerns the issue in question.

KPMG

Contents

KPMG

KPMG

KPMG

Purpose

The document defines a set of general software security coding practices, in a checklist format, that can be integrated into the software development lifecycle. Implementation of these practices will mitigate most common software vulnerabilities.The software development either for the clients or internal purpose should be done against these guidelines.

The owner has to ensure security testing of the software has to be done and all identified high and medium vulnerabilities to be addressed before making the software live in production.

Development Lifecycle

- Security must be a focus of attention at all stages of the systems development lifecycle. Whereas an individual project is bounded between requirements specification and deployment the secure systems development lifecycle spans the period between requirements specification and decommission.
- For the purpose of this standard no specific project or operational methodology has been assumed, instead the requirements of this standard have been mapped to a set of generic lifecycle stages.
- It is not intended that these lifecycle stages are adopted in preference to a functions systems development or systems maintenance processes. Rather the function should ensure that the stages within this lifecycle are mapped to KPMG India's own processes. This will ensure:
  - That the requirements of the secure systems development lifecycle are integrated into internal processes.
  - That governance of the secure systems development lifecycle is integrated into project governance processes and that these integration points are compatible with the various security standard.

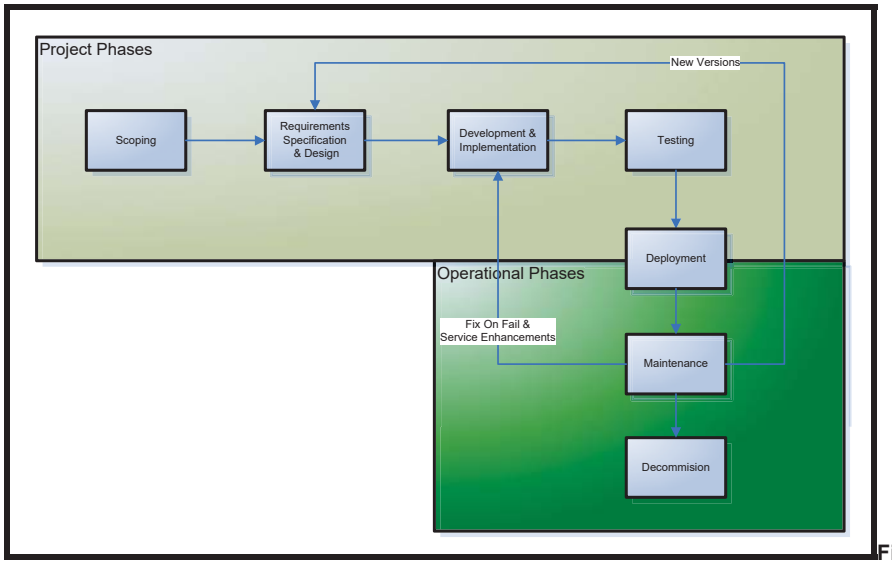


Figure 1: The Phases of the Secure Systems Development Lifecycle

This standard has defined the secure systems development lifecycle stages to be:

- Scoping: Where financial and resource planning activities take place and project governance defined.
- Requirements Specification & Design: Where the analysis and specification of business, technical and non-functional requirements is undertaken.
- Development & Implementation: Where the system is constructed, tested and approved for release into production.
- Testing: Where the system is tested against its requirements.
- Deployment: Where the system is accepted into production and transferred to systems maintenance and support teams.

- Maintenance: Where the system is supported and operated in the production environment.
- Decommission: Where the systems is no longer required and is decommissioned

1.1. Secure coding principle

1. Input Validation
2. Output Encoding
3. Authentication and Password Management (includes secure handling of credentials by external services/scripts)
4. Session Management
5. Access Control
6. Cryptographic Practices
7. Error Handling and Logging
8. Data Protection
9. Communication Security
10. System Configuration
11. Database Security
12. File Management
13. Memory Management
14. General Coding Practices

1.2 Secure Coding Practices

Secure coding practices must be incorporated into all life cycle stages of an application development process. The following minimum set of secure coding practices should be implemented when developing and deploying covered applications:

1. Formalize and document the software development life cycle (SDLC) processes to incorporate major component of a development process:
    - Requirements
    - Architecture and Design
    - Implementation
    - Testing
    - Deployment
    - Maintenance

Software Security and Risk Principles

Building secure software requires a basic understanding of security principles. The goal of software security is to maintain the ***confidentiality, integrity, and availability*** of information resources in order to enable successful business operations. This goal is accomplished through the implementation of security controls. This documents focuses on the technical controls specific to mitigating the occurrence of common software vulnerabilities. While the primary focus is web applications and their supporting infrastructure, most of the standard can be applied to any software deployment platform.

It is helpful to understand what is meant by risk, in order to protect the business from unacceptable risks associated with its reliance on software. Risk is a combination of factors that threaten the success of the business. This can be described conceptually as follows: a threat agent interacts with a system, which may have a vulnerability that can be exploited in order to cause an impact. All of these factors play a role in secure software development.

Software security flaws can be introduced at any stage of the software development lifecycle, including:

- Not identifying security requirements up front
- Creating conceptual designs that have logic errors
- Using poor coding practices that introduce technical vulnerabilities
- Deploying the software improperly
- Introducing flaws during maintenance or updating

Furthermore, it is important to understand that software vulnerabilities can have a scope beyond the software itself. Depending on the nature of the software, the vulnerability and the supporting infrastructure, the impacts of a successful exploitation can include compromises to any or all of the following:

- The software and its associated information
- The operating systems of the associated servers
- The backend database
- Other applications in a shared environment
- The user's system
- Other software that the user interacts with

2 Security Architecture and Design Principles

The design of the solution should be undertaken to ensure that the following security principles are adhered to. These principles are based on good practice, and their application will assist the designer and developer to produce a solution that delivers cost-effective security without needing to compromise the functionality that the solution delivers.

Validate Input and Output

All user-input and output should be checked to ensure it is expected and appropriate. The most appropriate (and simplest) strategy when validating data is to validate against what is acceptable and drop/ignore other data submitted. The majority of attacks can be prevented or significantly mitigated simply by using appropriate validation techniques. Validation is a fundamental design requirement for building secure applications. Failing to validate input to and output from applications will invariably result in the application being vulnerable to attack.

General Countermeasures

- Accept only known valid data
  This is often the simplest approach to validation. For example, resetting a customer's PIN would require ASCII 0-9. The application should check the input for this type of string comprises of 0-9 (performing canonicalization checks as appropriate) and is of a valid length.
- Reject known bad / malicious data
  Applying validation routines to identify specific malicious payloads. Whilst this method can limit exposure, it is a resource intensive strategy and very difficult to maintain an up to date list of attack signatures.
- Sanitise all data
  The process of removing elements of submitted data that could be of a malicious nature. This provides a useful second line of defence, but should not be used as the primary line of defence.
- Never rely on client-side data validation
  This form of validation is easily bypassed. All validation should be performed server-side, client-side validation should only be considered in addition to server-side (but never replace it).

Input Validation:

- Conduct all data validation on a trusted system (e.g., The server)
- Identify all data sources and classify them into trusted and untrusted. Validate all data from untrusted sources (e.g., Databases, file streams, etc.)
- There should be a centralized input validation routine for the application
- Specify proper character sets, such as UTF-8, for all sources of input
- Encode data to a common character set before validating (*Canonicalize*)
- All validation failures should result in input rejection
- Determine if the system supports UTF-8 extended character sets and if so, validate after UTF-8 decoding is completed
- 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
- Verify that header values in both requests and responses contain only ASCII characters
- 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)
- Validate for expected data types
- Validate data range
- Validate data length
- Validate all input against a "white" list of allowed characters, whenever possible
- 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:
- < > " ' % ( ) & + \ \' \"
- If your standard validation routine cannot address the following inputs, then they should be checked discretely
- Check for null bytes (%00)
- Check for new line characters (%0d, %0a, \r, \n)
- Check for “dot-dot-slash" (../ or ..\) path alterations characters. In cases where UTF-8 extended character set encoding is supported, address alternate representation like: %c0%ae%c0%ae/
- (Utilize *canonicalization* to address double encoding or other forms of obfuscation attacks)

Output Encoding:

- Conduct all encoding on a trusted system (e.g., The server)
- Utilize a standard, tested routine for each type of outbound encoding
- *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
- Encode all characters unless they are known to be safe for the intended interpreter
- Contextually *sanitize* all output of un-trusted data to queries for SQL, XML, and LDAP
- *Sanitize* all output of un-trusted data to operating system commands

Input/ Output Validation Threat Matrix

Threat	Description	Mitigating Controls / Countermeasures
Threat	**Description**	**Mitigating Controls / Countermeasures**
Cross-site Scripting	These attacks are exploited when an attacker uses a web application to submit malicious code, often JavaScript (but could be any embedded active content such as ActiveX, VBscript, Shockwave, Flash, etc), to another user. If an application takes user input and outputs it (for example, submitting messages to an http-based message service), an attacker can input malicious code which is then displayed back to other users who have access to the output. Attackers often try to conceal the malicious code by using encoding techniques such as Unicode.	Validate headers, cookies, query strings, form fields, etc against allowed parameters, content, etc. • Applications can gain considerable protection by converting the following characters in generated output (may languages have functions to do this automatically): **From** → **To** < → < > → > (→ (

Threat	Description	Mitigating Controls / Countermeasures
	There are two primary cross-site scripting categories: Stored Injected code is stored permanently in a database (for example, visitor log, message, etc) Reflected Injected code takes an alternative route to the victim such as e-mail. The threats range from trivial annoyance such as displaying unexpected content to redirecting the user to another site; to more serious cases of session hijack, unauthorised disclosure of content and web site content changes. Web servers, application servers and web application environments are at risk of cross-site scripting.	)) # # & &

Threat	Description	Mitigating Controls / Countermeasures
SQL Injection	Without vigorous validation of user input, attackers may be able to make direct database calls to the database. The common name for this type of attack is SQL injection and can be achieved surprisingly simply. Poorly validated parameters may contain SQL commands which when submitted to the application are extracted as is and placed into a dynamic database query, which is subsequently executed on the database under the privilege of the application account (where the risk increases with the level of privilege the account has). The consequences can be devastating, examples can include: ▪ Compromise to the privacy of Customer lists ▪ Customer personal details (financial, medical, etc) ▪ Inadvertent change to administrator or other customer passwords ▪ Unauthorised alteration of data and affecting the database integrity ▪ Loss of essential tables	Validate against acceptable input, drop all other input Construct all queries and prepared statements and parameterised stored procedures. A prepared statement or parameterised stored procedure encapsulates variables and should escape special characters within them automatically and in a manner suited to the targeted database. Never give an application DBADMIN privileges. The web application should run with the minimum privileges it requires to perform its function. Validate output and return codes to ensure the expected processing occurred. Ensure that the authenticated user's profile is authorised to submit queries to selected tables.

Threat	Description	Mitigating Controls / Countermeasures
S Command Injection	The majority of programming languages allow the use of system commands and many applications make use of this functionality. System interfaces in programming and scripting languages pass input commands to the underlying operating system. The operating system in turn executes this input and returns the output with various return codes to the application such as successful, not successful, etc. Depending on the programming or scripting language and the operating system, it is possible to: Alter system commands Alter parameters passed to system commands Execute additional commands and OS command-line tools Execute additional commands within executed commands **Some functions/statements to avoid:** Java (Servlets, JSPs) System.* (especially System.Runtime)	Validate against expected input, drop all other input Never allow web servers to run as ROOT or ADMINISTRATOR. The web application should run with the minimum privileges it requires to perform its function. If OS commands must be used, all information inserted into the command should be rigorously checked. Implement mechanisms to handle possible errors, timeouts or blockages during the request. Validate output and return codes to ensure the expected processing occurred. Restrict access to command files e.g., cmd.exe

Threat	Description	Mitigating Controls / Countermeasures
	C & C++ system( ) exec**() strcpy strcat sprintf vsprintf gets strlen scanf fscanf sscanf vscanf vsscanf vfscanf realpath getopt getpass streadd strecpy strtrns	
Path Traversal	It is possible to utilise the file system of a web server to temporarily or permanently store information. If applications and web servers do not properly validate and handle meta-characters used to describe paths (e.g. '../'), it is possible that an application is vulnerable to a path traversal attack. The attacker can construct a malicious request to return data about physical file locations such as /etc/passwd. This is sometimes referred to as a file disclosure vulnerability. Path traversal attacks are often used in conjunction with other attacks such as operating system commands and direct SQL injection.	Where at all possible make use of path normalisation functions provided by the development language. Remove offending path strings, such as '../' as well as their Unicode variants from system input. Use full directory paths. Validate against expected input, drop all other input.

Meta Characters	Non-printable and printable characters which affect the behaviour of programming language commands, operating system commands, program procedures and database queries. They are typically injected into URL-encoded parameters within query strings. Examples of Meta Characters: [;]Semicolons for additional command-execution [|]Pipes for command-execution [!]Call signs for command-execution [&]for command-execution [x20]Spaces for faking URLs and other names (especial in URLs!) [x00]Null bytes for cutting off strings and filenames [x04]EOT for faking file ends [x0a]New lines for additional command-execution, fake-mails/change file-content [x0d]New lines for additional command-execution, fake-mails/change file-content [x1b]Escape OS-dependent [x08]Backspace OS-dependent (faking log files, changing file-content) [x7f]Delete OS-dependent [~]Tildes OS-dependent (circumvent auth. on some ms-web servers) [' "]Quotation-marks in combination with database-queries [-]in combination with database-queries and	Remove meta-chars from user-input Check if given input has the expected data-type IIS URL Scan and intruder prevention (application shields) can be configured to block such characters.

Threat	Description	Mitigating Controls / Countermeasures
	creation of negative numbers [*%]in combination with database-queries [`]Backticks for command execution [/\ ]Slashes and Backslashes for faking paths and queries [<>]LTs and GTs for file-operations [<>]for creating script-language related TAGS within documents on web servers [?]programming/scripting- language related [$]programming/scripting- language related [@]programming/scripting- language related [:]programming/scripting- language related [({[]})]programming/scripting/regex and language-related	

Threat	Description	Mitigating Controls / Countermeasures
Null Bytes	Many programming applications often pass data to underlying low level C-functions for further processing and functionality. If a given string “XXX\0YYY” is accepted as valid by an application, it may be shortened to “XXX” by the underlying C-functions. This occurs because C/C++ perceives the null byte (\0) as the termination of the string. Applications which do not perform adequate input validation can be fooled by inserting null bytes in “critical” parameters. This is normally done by URL encoding the null bytes (%00). In special cases it is possible to use Unicode characters. This type of attack can: • Disclose physical paths, files and operating system information • Truncate strings • OS Command execution • Command parameters • Bypass validity checks, looking for sub-strings in parameters • Cut off strings passed to SQL queries	Validate all input before the application acts upon it.

Threat	Description	Mitigating Controls / Countermeasures
Buffer Overflow	The process involves the submitting of large amounts of data that exceed the quantities expected by the application for a given input field or query string parameter. Successful buffer overflow exploits can result in unexpected behaviour of the application, allowing the attacker to execute commands in the context of the application. The risk is exacerbated if the application is running under a system-level or administrative operating system account. Please note: *This is less of an issue when using 'managed languages' such as Visual Basic, PERL and C# as they have built in memory management functions which prevent buffer overflow conditions.*	▪ Validate input string and drop any requests that exceed the pre-defined string size. ▪ Validate URL query string, content and headers and drop any requests that exceed the pre-defined sizes set. ▪ Ensure tools such as URL Scan apply correct restrictions.

2.2 Canonicalization

Management Overview

Canonicalization is the process of converting something from one representation to the simplest form. Web applications have to deal with numerous canonicalization issues from URL encoding to IP address translation. Web applications need to be able to deal with canonicalisation issues accurately.

Canonicalisation Threat Matrix

Threat	Description	Mitigating Controls / Countermeasures
Unicode	Unicode is the internal format of Java. Unicode Encoding is a method for storing characters with multiple bytes. Wherever input data is permitted, Unicode can be entered to disguise malicious code and consequently permit a variety of attacks. RFC2279 references many ways in which text can be encoded.	• Select a suitable canonical form and ensure all user input is canonicalised into that form before any authorisation decisions are performed. • Security checks should be performed after decoding is complete. • Verify that UTF-8 encoding is a valid canonical encoding for the symbol it represents.
URL Encoding	Traditionally web applications transfer data between server and client using HTTP or HTTPS protocols. There are two primary methods in which a server receives input from the client: GET - Data is passed in the query section of a URL. POST - Data is passed in the HTTP headers. When data is included in a URL it needs to be encoded to conform to proper URL syntax. RFC1738 defines URLs and RFC2396 defines URIs, both restrict the characters permitted in a URL or URI to a subset of the US-ASCII character set. RFC1738 specifies: *"Only alphanumeric, special characters "$-_.+!*'()," and reserved characters used for the reserved purposes may be used un-encoded within a URL.".* However the data used by a web application is not restricted in this way. Earlier versions of HTML permitted the entire range of ISO-8859-1 [ISO Latin-1)	• Do not use HTTP GET to submit forms, use HTTP POST to avoid appending data to the URL. • If a URL has to be used to pass data to the web server, limit the type of data permitted and do not permit textual data. Employ strict validation rules to sanitise and ensure submitted data is the correct type and size • Do not rely on client-side validation.

Threat	Description	Mitigating Controls / Countermeasures
	character set. The HTML 4.0 specification has subsequently expanded further to permit any character in the Unicode set. To URL Encode a character, the 8-bit hexadecimal code is taken and prefixed with a % sign. An example is the US-ASCII character set represents a space with decimal code 32. Decimal code 32 translates to hexadecimal 20. Users of web applications will therefore see spaces replaced by %20 in the URL. Despite some characters not needing to be URL encoded, any 8bit code may be encoded. Examples of this: ASCII control characters such as NULL All HTML entities All meta characters used by the operating system or database As URL encoding permits virtually any data to be passed to the server, appropriate precautions must be taken by the web application. Without adequate precautions the application can be susceptible to malicious acts.	

1.2 Authetication and password management

- Require authentication for all pages and resources, except those specifically intended to be public
- All authentication controls must be enforced on a trusted system (e.g., The server)
- Establish and utilize standard, tested, authentication services whenever possible
- Use a centralized implementation for all authentication controls, including libraries that call external authentication services
- Segregate authentication logic from the resource being requested and use redirection to and from the centralized authentication control
- All authentication controls should fail securely
- All administrative and account management functions must be at least as secure as the primary authentication mechanism
- 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)
- Password hashing must be implemented on a trusted system (e.g., The server).
- Validate the authentication data only on completion of all data input, especially for *sequential authentication* implementations
- 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
- Utilize authentication for connections to external systems that involve sensitive information or functions
- 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
- Use only HTTP POST requests to transmit authentication credentials
- 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
- 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)
- 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
- Password entry should be obscured on the user's screen. (e.g., on web forms use the input type "password")

- 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
- Password reset and changing operations require the same level of controls as account creation and authentication.
- Password reset questions should support sufficiently random answers. (e.g., "favorite book" is a bad question because “The Bible” is a very common answer)
- If using email based resets, only send email to a pre-registered address with a temporary link/password
- Temporary passwords and links should have a short expiration time
- Enforce the changing of temporary passwords on the next use
- Notify users when a password reset occurs
- Prevent password re-use
- Passwords should be at least one day old before they can be changed, to prevent attacks on password re-use
- 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
- Disable "remember me" functionality for password fields
- The last use (successful or unsuccessful) of a user account should be reported to the user at their next successful login
- 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
- Change all vendor-supplied default passwords and user IDs or disable the associated accounts
- Re-authenticate users prior to performing critical operations
- Use *Multi-Factor Authentication* for highly sensitive or high value transactional accounts
- If using third party code for authentication, inspect the code carefully to ensure it is not affected by any malicious code

1.3 Session Management

- Use the server or framework’s session management controls. The application should only recognize these session identifiers as valid
- Session identifier creation must always be done on a trusted system (e.g., The server)
- Session management controls should use well vetted algorithms that ensure sufficiently random session identifiers
- Set the domain and path for cookies containing authenticated session identifiers to an appropriately restricted value for the site
- Logout functionality should fully terminate the associated session or connection

- Logout functionality should be available from all pages protected by authorization
- Establish a session inactivity timeout that is as short as possible, based on balancing risk and business functional requirements. In most cases it should be no more than several hours
- 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
- If a session was established before login, close that session and establish a new session after a successful login
- Generate a new session identifier on any re-authentication
- Do not allow concurrent logins with the same user ID
- Do not expose session identifiers in URLs, error messages or logs. Session identifiers should only be located in the HTTP cookie header. For example, do not pass session identifiers as GET parameters
- Protect server side session data from unauthorized access, by other users of the server, by implementing appropriate access controls on the server
- Generate a new session identifier and deactivate the old one periodically. (This can mitigate certain session hijacking scenarios where the original identifier was compromised)
- 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.
- 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
- Supplement standard session management for highly sensitive or critical operations by utilizing per-request, as opposed to per-session, strong random tokens or parameters
- Set the "secure" attribute for cookies transmitted over an TLS connection
- Set cookies with the HttpOnly attribute, unless you specifically require client-side scripts within your application to read or set a cookie's value

1.4 System Configuration

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

- Turn off directory listings
- Restrict the web server, process and service accounts to the least privileges possible
- When exceptions occur, fail securely
- Remove all unnecessary functionality and files
- Remove test code or any functionality not intended for production, prior to deployment
- Prevent disclosure of your directory structure in the robots.txt file by placing directories not intended for public indexing into an isolated parent directory. Then "Disallow" that entire parent directory in the robots.txt file rather than Disallowing each individual directory
- Define which HTTP methods, Get or Post, the application will support and whether it will be handled differently in different pages in the application
- 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
- 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)
- Remove unnecessary information from HTTP response headers related to the OS, web-server version and application frameworks
- The security configuration store for the application should be able to be output in human readable form to support auditing
- Implement an asset management system and register system components and software in it
- 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
- Implement a software change control system to manage and record changes to the code both in development and production

1.5 Parameter Manipulation & Communication security

Management Overview

Parameter manipulation is a relatively simple process and can be a very effective method of compromising web application security by making applications perform activities which they should not be permitted to do. Attackers may take advantage of poorly designed or written applications to change such things as prices of goods being ordered, altering session tokens or values stored in cookies or HTTP headers.

No data sent to the browser can be relied upon to remain the same unless cryptographically protected at the application layer. SSL will not protect against this type of attack as data is manipulated before it is sent to the server. This wowuld include following :

- 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
- TLS certificates should be valid and have the correct domain name, not be expired, and be installed with intermediate certificates when required
- Failed TLS connections should not fall back to an insecure connection
- Utilize TLS connections for all content requiring authenticated access and for all other sensitive information
- Utilize TLS for connections to external systems that involve sensitive information or functions
- Utilize a single standard TLS implementation that is configured appropriately
- Specify character encodings for all connections
- Filter parameters containing sensitive information from the HTTP referer, when linking to external sites

Parameter Manipulation Threat Matrix

Threat	Description	Mitigating Controls / Countermeasures
Cookie Manipulation	All forms of cookies may be tampered with prior to sending back to the server. The extent of cookie manipulation depends on the use the cookie is put to. Many cookies are Base64 encoded which does not offer any cryptographic protection.	• Do not trust user input for values that you already know. • Use one session token to reference properties stored on a server-side cache.

<table>
<tr><th>Threat</th><th>Description</th><th>Mitigating Controls / Countermeasures</th></tr>
<tr>
<td>Form Field Manipulation</td>
<td>Information selected or entered is typically stored as form field values and sent to the application via HTTP requests (GET or POST). HTML can also store field values as hidden fields, which are not rendered to the screen by the browser, but collected and submitted as parameters during form submissions.

Irrespective of the form field type (drop down, check or text boxes, etc), they can all be manipulated by the user to submit whatever values they choose. This can be easily achieved in most cases by viewing the source and saving the file for editing and subsequently re-loading the manipulated file.

Some examples of ineffective client-side form field manipulation:
<table>
<tr><th>Start Code</th><th>Altered Code</th><th>Results</th></tr>
<tr><td><input name=”debug” type=”hidden” value=”0”></td><td><input name=”debug” type=”hidden” value=”1”></td><td><u>Privilege Elevation</u>
Attacker changes the value from 0 to 1 to switch on debug mode which could enable additional application functionality, reveal system and database passwords, application logic, etc)</td></tr>
</table>
</td>
<td>Never rely on client-side validation. Always validate input server-side.
Avoid hidden fields, using one session token to reference data stored in a server-side cache. If an application needs to check a user property, it checks the session cookie with its session table and points the user’s data variables in the cache/database.
Where it is not possible to implement the above solution and the use of hidden fields is required, concatenate the name and value pairs together into a single string and append a secret key (which never appears in the form) to the end of the string. The string is called the outgoing form message, an MD5 or similar one-way hash is generated for the outgoing form message. This is called the outgoing form digest and is added to the form as an additional hidden field.
• When the form is submitted, the name and value pairs are concatenated once again along with the secret key into the incoming form message. An MD5 digest of the incoming form message is generated (submitted with the form) and if they do not match, the hidden field must have been tampered with. The same technique can</td>
</tr>
</table>

Threat	Description			Mitigating Controls / Countermeasures
	<input name=”userid” type=”hidden” maxlength=”10”>	<input name=”userid” type=”hidden”>	Buffer Overflow Attacker removes the maxlength tag to remove the client-side limit of 10 characters in the user id field and proceed to attempt buffer overflow exploits	be applied to URLs to prevent manipulation of parameters.
	<input name=”adminaccess” type=”hidden” value=”n”>	<input name=”adminaccess” type=”hidden” value=”y”>	Privilege Elevation Attacker changes the value from ‘n’ to ‘y’, causing the application to elevate their privilege to administrator level access.	

Threat	Description	Mitigating Controls / Countermeasures
HTTP Header Manipulation	HTTP headers are used to pass control information from web clients to web servers for HTTP requests and vice versa for HTTP responses. Each header normally consists of a single line of ASCII text with a name and value.	Do not rely on headers without additional security mechanisms.

Threat	Description	Mitigating Controls / Countermeasures
	Generally HTTP headers are only used by the browser and web server; the majority of applications ignore them. Web developers do sometimes choose to inspect incoming headers and in such cases it is important to understand that as they originate from the client, they can be altered by an attacker, Examples: ▪ Referer header. Normally contains the URL of the page which the request originated. Developer's may choose to check this header in order to confirm the request originated from a trusted URL (e.g. their own) in the belief it prevents attackers from saving web pages, modifying the forms and posting them from a different computer. This is not a security mechanism as the attacker will be able to modify the HTTP Referer header to appear to have come from the trusted site. Accept language header: Indicates the preferred language/s of the user. An application may select the language label from the HTTP header and pass it to a database in order to look up text. If the content of the header is sent verbatim to the database, an attacker may be able to inject SQL commands by modifying the header. Similarly, if the header is used to build a name of a file from which to look up the correct language text,	

Threat	Description	Mitigating Controls / Countermeasures
	an attacker may be able to launch a path traversal attack.	

URL Manipulation	HTML forms can submit their results using two methods, HTTP POST and HTTP GET. If HTTP GET is used, all form element names and values will appear in the URL query string, enabling the attacker to easily tamper with the values, attempt to submit unexpected data, etc.	• Avoid the use of parameters in the query string.
		• Where parameters need to be submitted to a server, ensure they are accompanied with a valid session token.
		• If a parameter cannot be removed from the query string, it should be cryptographically protected using approved and strong cryptographic algorithms. The methods for achieving this are:
		  • Encryption of the complete query string
		  • Add an extra parameter in the query string. The value, an MD5 digest of the query string. It doesn't prevent the user from viewing the string, but if the application checks the returned hash and fails requests where hashes do not match, it prevents them from successfully changing and submitting it.

1.6 Information Disclosure and User Privacy

Management Overview

Attackers use a number of methods to obtain information that could provide a useful basis on which to perpetrate an attack on web sites or the supporting infrastructures. However it is possible, in many areas, to prevent this form of unwanted disclosure by following simple guidelines.

Information Disclosure and User Privacy Threat Matrix

Threat	Description	Mitigating Controls / Countermeasures
Client-side Comments	Adding and maintaining comments in source code has been a standard developer practice for years to assist in later support. This practice often extends to HTML pages which, depending on the nature of the comments, can reveal sensitive information about the structure of the web site, its underlying infrastructure or members of staff. Comments which are often left in HTML pages include directory structures, server names, IP addresses, errors, debug information, developer names, change/problem numbers, email addresses, phone numbers.	▪ Remove comments (except Copyright comments!) from code before it is migrated to production services. ▪ Ensure quality assurance processes include provision to verify that all comments are removed prior to migration to production.
Debug Commands	As with source code comments, it is often standard developer practice to include debug switches in HTML to allow them to switch on additional levels of logging or reporting. Permitting this code (and the server-side logic to interpret it) into production services result in a serious vulnerability, gaining the attacker escalated privileges to the service and underlying infrastructure.	▪ Implement a standard debugging process, using specific tools and practices making it easier to remove this functionality when required. ▪ Consider automating debug function removal and ensure all debug functionality is removed prior to migration to production.

Threat	Description	Mitigating Controls / Countermeasures
		▪ Prior to migration to production perform tests to ensure the server-side debug logic has been removed.
Error Codes	Poorly implemented error code handling can provide an attacker the insight they require to launch an attack on a web application or supporting infrastructure. The types of information available to the attacker might include: ▪ Flow of application ▪ Additional web server information ▪ Database type and version ▪ Operating system type and version ▪ Scripting/programming language type and version ▪ Physical paths ▪ Files opened for reading ▪ Files opened for writing ▪ Names and values of variables ▪ Types of variables ▪ Purpose of variables ▪ Segments of script source code ▪ Segments of SQL Queries ▪ Database and table structures	▪ Where possible, avoid reporting error messages to the user in production systems. In circumstances where errors cannot be prevented, ensure the errors are suitably masked and do not reveal potentially use information to an attacker. ▪ Ensure sufficient logging is available to capture errors for internal support purposes.
File/Application Enumeration	This is a common technique used to identify files and applications that could be vulnerable to exploitation or provide a foundation on which to form an attack. Attackers look for: ▪ Vulnerable files or applications ▪ Hidden or un-referenced files or applications Backup or temporary files	Remove all sample files from web servers Remove unwanted or unused files from servers Regular search for and remove backup or temporary files

Browser Cache	Often sensitive information can be stored in the browser cache, which could be accessed by anybody with access to the machine's hard drive storing the browser cache. Typical examples could be office PCs, Internet cafés, etc.	▪ Applications should only serve sensitive information to the intended recipient when absolutely necessary. ▪ Where possible pre-expire pages likely to hold sensitive material. ▪ Include the 'Pragma No-Cache' command on all pages likely to hold sensitive material, which instructs browsers not to cache the pages.
Browser History	Browsers often retain a history of recently visited web sites that are offered by the browser when a user begins to enter a similar URL. URLs can often contain parameters that could be used to reveal enough information on which to base an attack. The parameters included on the URLs can be re-submitted if the method of transferring the data is HTTP GET. Data transferred using HTTP POST is not appended on the URL and therefore not saved.	▪ Use HTTP POST to submit form data, never use HTTP GET.
AutoComplete	Microsoft Internet Explorer 5 and above supports the AutoComplete function. This can be a useful feature whereby users' input can be stored for future use and presented to the user of the computer when they	Ensure customers are warned about the feature and recommend they disable it if using shared machines.

	click on a web form field with the same name. Netscape support a similar function called Password Manager and Form Manager. Refer to the Microsoft description of the feature and details of how to configure and disable it. If this feature is enabled on shared computers (such as offices, libraries, Internet Cafes), information which customers enter into input fields on one web-site may be visible by other users who use the computer afterwards. This could include personal and financial details Norwich Union would not be responsible under the Data Protection Act 1998 as the information is stored on the client machine. However, as this feature could present a risk, KPMG India should consider the potential risks and benefits before deciding what to do.	Inform customers they leave this function enabled on shared machines at their own risk. • Disable the function on password/PIN fields • Disable the function on fields such as bank card and account details Depending on the nature of the business, consider disabling the function entirely
Password Storage and hard coded passwords	A database storing passwords if compromised could result in a serious breach of security. Account details stored in .asp, .php, .dll files (for example) have in the past been compromised when bugs in web servers have allowed source of specific files to be viewed by the browser.	▪ Where possible, avoid storing passwords, PINs, etc. Instead store a hash of the password using a one-way hashing algorithm or message digest. Use forms authentication (GAS) to avoid browsers from saving the password/PIN, etc. In circumstances where passwords, PINs etc must be stored, ensure they are encrypted

		using a strong and approved encryption algorithm.
User Education	Not everybody who has access to the Internet is a computer security expert and many do not understand why it is important to them.	▪ Provide considered advice approved by the Norwich Union security, public relations and marketing communities. Where possible use existing advice currently available on the corporate web-sites.
Hidden Fields	Hidden fields can be useful, but can present significant risks to an application if used inappropriately to store sensitive information. Hidden fields can be easily seen, modified and re-sent by an attacker with limited experience.	▪ Values which could be used by an attacker to effect an unexpected response (perhaps obtain another person's data or generate an error condition which could form the basis of an attack) should always be encrypted or hashed ▪ Avoid storing session ids in these fields ▪ Never store passwords or PINs in hidden fields ▪ Any personal data (as defined by the Data Protection Act 1998) or financial information should be encrypted and sent across SSL encrypted sessions ▪ These fields should always be rigorously validated using server-side validation only ▪ Never use hidden fields for web server control commands
Account History	Generally users of applications have no way of verifying whether their account has been accessed/mis-used by unauthorised parties.	▪ Consider introducing a 'last login' time, date and source IP address to be displayed after they successfully authenticate. ▪ Consider building the provision of a more detailed account history section for

		authenticated users, including recent successful authentications (time and date stamped), modifications to the account such as password changes (time and date stamped), financial transactions, etc.
Incident Reporting	If a user notices suspicious behaviour changes to their account or content of the site, it is important they know how to report the matter to the company. Without clear and simple instructions, there is a risk the issue may not be reported.	▪ Ensure the applications/hosting encourages users to report issues, along with a documented process approved by relevant security departments.
Sensitive Information and Source Code	Client-side source code is easily visible to users, inserting hard coded sensitive information into source code could provide an attacker with the information they require to mount an attack or to commit a fraud.	▪ Do not hard code sensitive information into client-side code, such as IDs, passwords, etc.
Sensitive Information and Cookies	Cookies can be viewed and modified. If cookies store sensitive information, this information could be used to mount an attack or to commit a fraud.	▪ Do not store personal data (as defined by the Data Protection Act 1998) or financial information in cookies. ▪ Do not store authentication details in cookies. ▪ Ensure session IDs are hashed if stored in cookies. ▪ Encrypt contents of cookies using approved crypto APIs. ▪ Consider the use of the 'Secure' label to prevent browsers sending the cookie over an unencrypted connection.
Cryptography	Employing cryptography is not a complete solution to data security. Cryptography can be employed to provide: ▪ Confidentiality (data is understood by authorised parties only)	When implementing encryption it is vital to: ▪ Understand the business and security requirements

	▪ Integrity (data is not altered in transit) ▪ Authentication (data originates from a specific party) In reality cryptography is a challenging control to implement. Issues range from: ▪ False sense of security ▪ In-house developed and untested encryption routines ▪ Use of un-trusted and unsupported encryption routines ▪ System performance ▪ System/data recovery ▪ Key management and recovery ▪ Algorithm type/strengths ▪ Key lengths Key/random number generation	▪ Work closely with the business information and UK Security technical teams ▪ Do not attempt to develop an encryption routine yourself ▪ Do not use un-trusted or unsupported encryption routines ▪ Use only strong, approved and supported encryption routines ▪ Document the solution ▪ Test the solution thoroughly (encrypt, decrypt, recover) ▪ Ensure the key management system (manual or IT-based) is tested thoroughly and adequate training provided to support staff. The service must be capable of retrieving encrypted data for criminal investigations. Apply suitable key lengths

1.7 Access Control

Management Overview

Implementing adequate access control to protect front and back-end systems and data is essential. Access controls should be used to:

- Restrict what users can do
- Restrict which resources users have access to
- What functions they are permitted to perform on the data

Principally, access control mechanisms should prevent unauthorised viewing, modification or copying of data. In addition such mechanisms can also be employed to limit malicious code execution and unauthorised actions through an attacker exploiting infrastructure dependencies.

- Use only trusted system objects, e.g. server side session objects, for making access authorization decisions
- Use a single site-wide component to check access authorization. This includes libraries that call external authorization services
- Access controls should fail securely
- Deny all access if the application cannot access its security configuration information
- 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
- Segregate privileged logic from other application code
- Restrict access to files or other resources, including those outside the application's direct control, to only authorized users
- Restrict access to protected URLs to only authorized users
- Restrict access to protected functions to only authorized users
- Restrict direct object references to only authorized users
- Restrict access to services to only authorized users
- Restrict access to application data to only authorized users
- Restrict access to user and data attributes and policy information used by access controls
- Restrict access security-relevant configuration information to only authorized users
- Server side implementation and presentation layer representations of access control rules must match
- If *state data* must be stored on the client, use encryption and integrity checking on the server side to catch state tampering.
- Enforce application logic flows to comply with business rules
- 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
- Use the "referer" header as a supplemental check only, it should never be the sole authorization check, as it is can be spoofed
- 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
- 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.)
- The application must support disabling of accounts and terminating sessions when authorization ceases (e.g., Changes to role, employment status, business process, etc.)
- Service accounts or accounts supporting connections to or from external systems should have the least privilege possible

- 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

Access Control Threat Matrix

Threat	Description	Mitigating Controls / Countermeasures
Data Classification and Access Authorisation	If data is inappropriately classified or not classified at all, inappropriate access controls could be applied to data, resulting in unauthorized disclosure. Without an effective authorization process in place, access to data or systems may be granted inappropriately without the data or system owner's knowledge.	• Ensure all data used by the application is classified according to the Information Asset classification & protection standard. • Ensure an authorization process is in place, documented and reviewed regularly.
Unexpected resource access	Attackers will not always use an application in the manner in which It is expected to operate. Attackers will try to bypass the intended application controls to gain access to resources, processes or data which is ordinarily masked by the application logic.	• Identify and document roles and access privileges. • Follow principle of least privilege. • Make sure every protected resource authenticates the user session before permitting access. For example, if a user is submitting a query through the application, in addition to adequate input validation, the process should check that the user profile being used has authority to query the selected SQL table.

Threat	Description	Mitigating Controls / Countermeasures
Obscured vulnerability or data exploited through lack of adequate access control	Trying to protect critical or sensitive resources, processes or data through simple techniques such as file naming conventions, hidden files and folders, etc does not prevent attackers from accessing them unless supported by additional authentication and access control. The majority of professional attackers will use techniques to reveal such resources, etc over time.	• Always apply appropriate access controls over resources, processes and data.

1.8 Event Logging

Management Overview

Event logging is the method of recording an event performed by a user or system that can be reviewed and analysed at a later date. Event logging can be used to analyse a system problem or security compromise. Logging can:

- Provide indications of suspicious activities.
- Provide user accountability by tracking user actions.
- Be used to reconstruct events after a problem or exploit has occurred.
- Can (if handled appropriately) be a useful contribution in legal proceedings.

Error Handling and Logging:

- Do not disclose sensitive information in error responses, including system details, session identifiers or account information
- Use error handlers that do not display debugging or stack trace information
- Implement generic error messages and use custom error pages
- The application should handle application errors and not rely on the server configuration
- Properly free allocated memory when error conditions occur
- Error handling logic associated with security controls should deny access by default

- All logging controls should be implemented on a trusted system (e.g., The server)
- Logging controls should support both success and failure of specified security events
- Ensure logs contain important *log event data*
- Ensure log entries that include un-trusted data will not execute as code in the intended log viewing interface or software
- Restrict access to logs to only authorized individuals
- Utilize a master routine for all logging operations
- Do not store sensitive information in logs, including unnecessary system details, session identifiers or passwords
- Ensure that a mechanism exists to conduct log analysis
- Log all input validation failures
- Log all authentication attempts, especially failures
- Log all access control failures
- Log all apparent tampering events, including unexpected changes to state data
- Log attempts to connect with invalid or expired session tokens
- Log all system exceptions
- Log all administrative functions, including changes to the security configuration settings
- Log all backend TLS connection failures
- Log cryptographic module failures
- Use a cryptographic hash function to validate log entry integrity

Event Logging Threat Matrix

Threat	Description	Mitigating Controls / Countermeasures
Unable to detect and assess impact of a security compromise	Failing to design adequate event logging mechanisms into applications could undermine our ability to detect unauthorised activities and establish the consequential impact to our	When designing event logging, recording of the following information should be considered where appropriate: • Log files must be classified following the KPMG Classification Framework • Authentication attempts e.g. log in/out, failed log in attempts. • Authorisation attempts, including time, success/failure, resource or function being authorised and the user requesting authorisation.

Threat	Description	Mitigating Controls / Countermeasures
	systems, data or business interests.	• All administrative functions, such as account management, viewing user data, enabling or disabling event logging, etc. • Logging debug information must not result in user's private account credentials (e.g. password/PIN) being recorded in event logs. • Logs shall not be overwritten until a copy of the log has been archived or backed up. Archived and backup logs shall be protected and retained in-line with their classifications and purpose. • Contents of logs should only be accessible by those with a business need and appropriate authorisation from the system or data owner whose system or database is being monitored. • Log files shall only be updated by the event logging service, i.e. individual users shall not have the ability to update or delete items within the event logs. • Network communications • Event logging should be compatible with MQ Series and be capable of alerting critical events to specific individuals or devices. • Where risk assessment identifies the need for stringent non-repudiation of actions performed by an individual or entity, the logs and mechanisms designed to deliver them shall be to a standard which is acceptable as irrefutable evidence in a court of law.

1.9 Database security:

- Use strongly typed *parameterized queries*
- Utilize input validation and output encoding and be sure to address meta characters. If these fail, do not run the database command
- Ensure that variables are strongly typed
- The application should use the lowest possible level of privilege when accessing the database
- Use secure credentials for database access

- 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.
- Use stored procedures to abstract data access and allow for the removal of permissions to the base tables in the database
- Close the connection as soon as possible
- Remove or change all default database administrative passwords. Utilize strong passwords/phrases or implement multi-factor authentication
- 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))
- Remove unnecessary default vendor content (e.g., sample schemas)
- Disable any default accounts that are not required to support business requirements
- The application should connect to the database with different credentials for every trust distinction (e.g., user, read-only user, guest, administrators

1.10 File management:

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

1.11 Memory Management:

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

1.12 Cryptograpihic practice:

- All cryptographic functions used to protect secrets from the application user must be implemented on a trusted system (e.g., The server)
- Protect master secrets from unauthorized access
- Cryptographic modules should fail securely
- 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
- Establish and utilize a policy and process for how cryptographic keys will be managed

1.13 Data protection

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

- 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
- Protect server-side source-code from being downloaded by a user
- 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 viewstate, Adobe flash or compiled code
- Remove comments in user accessible production code that may reveal backend system or other sensitive information
- Remove unnecessary application and system documentation as this can reveal useful information to attackers
- Do not include sensitive information in HTTP GET request parameters
- Disable auto complete features on forms expected to contain sensitive information, including authentication
- Disable client side caching on pages containing sensitive information. Cache-Control: no-store, may be used in conjunction with the HTTP header control "Pragma: no-cache", which is less effective, but is HTTP/1.0 backward compatible
- The application should support the removal of sensitive data when that data is no longer required. (e.g. personal information or certain financial data)
- 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

1.14 Legal

Management Overview

There are a number of laws and Acts that staff and contractors working on behalf of the company are expected to understand and follow. Below are three typical Acts that have specific relevance to building and developing a new application or service.

Legal Threat Matrix

Threat	Description	Mitigating Controls / Countermeasures
Copyright, Designs and Patents Act 1988	The term computer program/software is defined as a Literary Work in the Copyright, Designs and Patents act (1988). When a user purchases a	• Purchase all programs/software using the NUCS PASC process. • Do not illegally copy software

	product they are only buying the rights to use the software in line with the terms and conditions within the licence agreement. Copyright is a property right, it gives the copyright owner the exclusive rights to produce copies, control or perform an original literary, musical, dramatic or artistic work. The Act allows the making of back-up copies of software, but only providing it is for lawful use. If there is any doubt over what constitutes a back-up, check your software licence agreement with the software publisher	• Refer to the Act for further information. *Pay attention to freeware products, whilst the license may permit free distribution and use, it may also require users of the freeware to publish any work they produce using the software.*
Data Protection Act 1998	This Act replaces the 1984 Act and consists of 8 "data protection principles" • Personal data shall be processed fairly and lawfully • Personal data shall be obtained and processed only for specified and lawful purposes • Personal data shall be adequate, relevant and not excessive in relation to the purpose of purposes for which they are produced • Personal data shall be accurate • Personal data must not kept longer than necessary	• Always ensure personal information is only accessible by those with authority to do so (for example, robust authentication mechanisms, access controls, etc) • Ensure the data is only used for the purposes which is has been registered. • Do not test software using real client data unless absolutely necessary. Where no alternative exists (i.e. de-sensitised or test data), ensure a formal request is sent to the data owner. • Refer to your local Data Protection contact for further advice.

	• Personal data shall be processed in accordance with the data subject's under this act • Personal data shall be secure • Personal data shall not be transferred to countries without adequate protection	
Computer Misuse Act 1990	The Act came into force on the 29th of August 1990, and an offence cannot be prosecuted under this Act unless every part of the offence was committed after this date. The Act defines three offences: • Unauthorised access to computer material • Unauthorised access with intent to commit or facilitate the commission of further offences • Unauthorised modification of computer material	• Do not access material and software code that you have no business justification to access. • Report any suspicious activity through the appropriate channels. • Do not alter material and software code that you are not permitted to change.

2 General Coding practice:

- Use tested and approved managed code rather than creating new unmanaged code for common tasks
- 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 through the use of application initiated command shells
- Use checksums or hashes to verify the integrity of interpreted code, libraries, executables, and configuration files
- Utilize locking to prevent multiple simultaneous requests or use a synchronization mechanism to prevent race conditions
- Protect shared variables and resources from inappropriate concurrent access
- Explicitly initialize all your variables and other data stores, either during declaration or just before the first usage
- In cases where the application must run with elevated privileges, raise privileges as late as possible, and drop them as soon as possible
- Avoid calculation errors by understanding your programming language's underlying representation and how it interacts with numeric calculation. Pay close attention to byte size discrepancies, precision, signed/unsigned distinctions, truncation, conversion and casting between types, "not-a-number" calculations, and how your language handles numbers that are too large or too small for its underlying representation
- Do not pass user supplied data to any dynamic execution function
- Restrict users from generating new code or altering existing code
- Review all secondary applications, third party code and libraries to determine business necessity and validate safe functionality, as these can introduce new vulnerabilities
- 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
- Do not use OS other then approved by KPMG global; In case of business requirement, licensed version to be used post approval from NITSO team.

3 Appendix: OWASP Top Ten

OWASP publish a top 10 of the most common web application security vulnerabilities with the aim to help educate developers, designers, architects and organizations about the consequences. This list was updated in 2017.

OWASP Top 10 Application Security Risks - 2017
A1-Injection
Injection flaws, such as SQL, OS, XXE, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
A2-Broken Authentication and Session Management

Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities (temporarily or permanently).
A3-Cross-Site Scripting (XSS)
XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user supplied data using a browser API that can create JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
A4-Broken Access Control
Restrictions on what authenticated users are allowed to do are not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.
A5-Security Misconfiguration
Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, platform, etc. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date.
A6-Sensitive Data Exposure
Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.
A7-Insufficient Attack Protection
The majority of applications and APIs lack the basic ability to detect, prevent, and respond to both manual and automated attacks. Attack protection goes far beyond basic input validation and involves automatically detecting, logging, responding, and even blocking exploit attempts. Application owners also need to be able to deploy patches quickly to protect against attacks.
A8-Cross-Site Request Forgery (CSRF)
A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. Such an attack allows the attacker to force a victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.
A9-Using Components with Known Vulnerabilities

Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
A10-Underprotected APIs
Modern applications often involve rich client applications and APIs, such as JavaScript in the browser and mobile apps that connect to an API of some kind (SOAP/XML, REST/JSON, RPC, GWT, etc.). These APIs are often unprotected and contain numerous vulnerabilities.

Note : 1. OWASP material has been used in the construction of this standard.

4 Refernce from OWASP Secure Coding Practices Quick Reference Guide

For any question regarding this procedure, please contact;

Group	**Management Information Security Forum**
Email	**In-fmnitso@kpmg.com**

Thank you