OWASP Secure Coding Practices Quick Reference Guide

Copyright and License

Copyright © 2010 The OWASP Foundation.

This document is released under the Creative Commons Attribution ShareAlike 3.0 license. For any reuse or distribution, you must make clear to others the license terms of this work.
http://creativecommons.org/licenses/by-sa/3.0/

Table of Contents

Introduction

This technology agnostic 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.

Generally, it is much less expensive to build secure software than to correct security issues after the software package has been completed, not to mention the costs that may be associated with a security breach.

Securing critical software resources is more important than ever as the focus of attackers has steadily moved toward the application layer. A 2009 SANS study[1] found that attacks against web applications constitute more than 60% of the total attack attempts observed on the Internet.

When utilizing this guide, development teams should start by assessing the maturity of their secure software development lifecycle and the knowledge level of their development staff. Since this guide does not cover the details of how to implement each coding practice, developers will either need to have the prior knowledge or have sufficient resources available that provide the necessary guidance. This guide provides coding practices that can be translated into coding requirements without the need for the developer to have an in depth understanding of security vulnerabilities and exploits. However, other members of the development team should have the responsibility, adequate training, tools and resources to validate that the design and implementation of the entire system is secure.

A glossary of important terms in this document, including section headings and words shown in *italics*, is provided in appendix B.

Guidance on implementing a secure software development framework is beyond the scope of this paper, however the following additional general practices and resources are recommended:

- Clearly define roles and responsibilities
- Provide development teams with adequate software security training
- Implement a secure software development lifecycle
  - OWASP CLASP Project
- Establish secure coding standards
  - OWASP Development Guide Project
- Build a re-usable object library
  - OWASP Enterprise Security API (ESAPI) Project
- Verify the effectiveness of security controls
  - OWASP Application Security Verification Standard (ASVS) Project)
- Establish secure outsourced development practices including defining security requirements and verification methodologies in both the request for proposal (RFP) and contract.
  - OWASP Legal Project

Software Security and Risk Principles Overview

Building secure software requires a basic understanding of security principles. While a comprehensive review of security principles is beyond the scope of this guide, a quick overview is provided.

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 guide 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 guidance 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*. While this may seem like an abstract concept, think of it this way: a car burglar (threat agent) goes through a parking lot checking cars (the system) for unlocked doors (the vulnerability) and when they find one, they open the door (the exploit) and take whatever is inside (the impact). All of these factors play a role in secure software development.

There is a fundamental difference between the approach taken by a development team and that taken by someone attacking an application. A development team typically approaches an application based on what it is intended to do. In other words, they are designing an application to perform specific tasks based on documented functional requirements and use cases. An attacker, on the other hand, is more interested in what an application can be made to do and operates on the principle that "any action not specifically denied, is allowed". To address this, some additional elements need to be integrated into the early stages of the software lifecycle. These new elements are *security requirements* and *abuse cases*. This guide is designed to help with identifying high level security requirements and addressing many common abuse scenarios.

It is important for web development teams to understand that client side controls like client based input validation, hidden fields and interface controls (e.g., pull downs and radio buttons), provide little if any security benefit. An attacker can use tools like client side web proxies (e.g. OWASP WebScarab, Burp) or network packet capture tools (e.g., WireShark) to analyze application traffic and submit custom built requests, bypassing the interface all together. Additionally, Flash, Java Applets and other client side objects can be decompiled and analyzed for flaws.

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

Secure Coding Practices Checklist

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

Authentication 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

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 HHTPS 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

Access Control:

- 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

Cryptographic Practices:

- 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
- 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)
- Establish and utilize a policy and process for how cryptographic keys will be managed

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

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

Communication Security:

- 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

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

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)

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 the chrooted 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

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

General Coding Practices:

- 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

Appendix A:

External References:

1. Cited Reference
   Sans and TippingPoint "The Top Cyber Security Risks"
   http://www.sans.org/top-cyber-security-risks/

- Web Application Security Consortium
  http://www.webappsec.org/

- Common Weakness Enumeration (CWE)
  http://cwe.mitre.org/

- Department of Homeland Security
  Build Security In Portal
  https://buildsecurityin.us-cert.gov/daisy/bsi/home.html

- CERT Secure Coding
  http://www.cert.org/secure-coding/

- MSDN Security Developer Center
  http://msdn.microsoft.com/en-us/security/default.aspx

- SQL Injection Cheat Sheet
  http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/

- Cross Site Scripting (XSS) Cheat Sheet
  http://ha.ckers.org/xss.html

Security Advisory Sites:

Useful resources to check for known vulnerabilities against supporting infrastructure and frameworks

Secunia Citrix Vulnerability List:
- http://secunia.com/advisories/search/?search=citrix

Security Focus Vulnerability Search:
- http://www.securityfocus.com/vulnerabilities

Open Source Vulnerability Database (OSVDB):
- http://osvdb.org/search/web_vuln_search

Common Vulnerability Enumeration:
- http://www.cve.mitre.org/

Appendix B: Glossary

Abuse Case: Describes the intentional and unintentional misuses of the software. Abuse cases should challenge the assumptions of the system design.

Access Control: A set of controls that grant or deny a user, or other entity, access to a system resource. This is usually based on hierarchical roles and individual privileges within a role, but also includes system to system interactions.

Authentication: A set of controls that are used to verify the identity of a user, or other entity, interacting with the software.

Availability: A measure of a system's accessibility and usability.

Canonicalize: To reduce various encodings and representations of data to a single simple form.

Communication Security: A set of controls that help ensure the software handles the sending and receiving of information in a secure manner.

Confidentiality: To ensure that information is disclosed only to authorized parties.

Contextual Output Encoding: Encoding output data based on how it will be utilized by the application. The specific methods vary depending on the way the output data is used. If the data is to be included in the response to the client, account for inclusion scenarios like: the body of an HTML document, an HTML attribute, within JavaScript, within a CSS or in a URL. You must also account for other use cases like SQL queries, XML and LDAP.

Cross Site Request Forgery: An external website or application forces a client to make an unintended request to another application that the client has an active session with. Applications are vulnerable when they use known, or predictable, URLs and parameters; and when the browser automatically transmits all required session information with each request to the vulnerable application. (This is one of the only attacks specifically discussed in this document and is only included because the associated vulnerability is very common and poorly understood.)

Cryptographic Practices: A set of controls that ensure cryptographic operations within the application are handled securely.

Data Protection: A set of controls that help ensure the software handles the storing of information in a secure manner.

Database Security: A set of controls that ensure that software interacts with a database in a secure manner and that the database is configured securely.

Error Handling and Logging: A set of practices that ensure the application handles errors safely and conducts proper event logging.

Exploit: To take advantage of a vulnerability. Typically this is an intentional action designed to compromise the software's security controls by leveraging a vulnerability.

File Management: A set of controls that cover the interaction between the code and other system files.

General Coding Practices: A set of controls that cover coding practices that do not fit easily into other categories.

Hazardous Character: Any character or encoded representation of a character that can effect the intended operation of the application or associated system by being interpreted to have a special meaning, outside the intended use of the character. These characters may be used to:

- Altering the structure of existing code or statements
- Inserting new unintended code
- Altering paths
- Causing unexpected outcomes from program functions or routines
- Causing error conditions
- Having any of the above effects on down stream applications or systems

HTML Entity Encode: The process of replacing certain ASCII characters with their HTML entity equivalents. For example, encoding would replace the less than character "<" with the HTML equivalent "<". HTML entities are 'inert' in most interpreters, especially browsers, which can mitigate certain client side attacks.

Impact: A measure of the negative effect to the business that results from the occurrence of an undesired event; what would be the result of a vulnerability being exploited.

Input Validation: A set of controls that verify the properties of all input data matches what is expected by the application including types, lengths, ranges, acceptable character sets and does not include known hazardous characters.

Integrity: The assurance that information is accurate, complete and valid, and has not been altered by an unauthorized action.

Log Event Data: This should include the following:

1. Time stamp from a trusted system component
2. Severity rating for each event
3. Tagging of security relevant events, if they are mixed with other log entries
4. Identity of the account/user that caused the event
5. Source IP address associated with the request
6. Event outcome (success or failure)
7. Description of the event

Memory Management: A set of controls that address memory and buffer usage.

Mitigate: Steps taken to reduce the severity of a vulnerability. These can include removing a vulnerability, making a vulnerability more difficult to exploit, or reducing the negative impact of a successful exploitation.

Multi-Factor Authentication: An authentication process that requires the user to produce multiple distinct types of credentials. Typically this is based on something they have (e.g., smartcard), something they know (e.g., a pin), or something they are (e.g., data from a biometric reader).

Output Encoding: A set of controls addressing the use of encoding to ensure data output by the application is safe.

Parameterized Queries (prepared statements): Keeps the query and data separate through the use of placeholders. The query structure is defined with place holders, the SQL statement is sent to the database and prepared, and then the prepared statement is combined with the parameter values. The prevents the query

from being altered, because the parameter values are combined with the compiled statement, not a SQL string.

Sanitize Data: The process of making potentially harmful data safe through the use of data removal, replacement, encoding or escaping of the characters.

Security Controls: An action that mitigates a potential vulnerability and helps ensure that the software behaves only in the expected manner.

Security Requirements: A set of design and functional requirements that help ensure the software is built and deployed in a secure manner.

Sequential Authentication: When authentication data is requested on successive pages rather than being requested all at once on a single page.

Session Management: A set of controls that help ensure web applications handle HTTP sessions in a secure manner.

State Data: When data or parameters are used, by the application or server, to emulate a persistent connection or track a client's status across a multi-request process or transaction.

System: A generic term covering the operating systems, web server, application frameworks and related infrastructure.

System Configuration: A set of controls that help ensure the infrastructure components supporting the software are deployed securely.

Threat Agent: Any entity which may have a negative impact on the system. This may be a malicious user who wants to compromise the system's security controls; however, it could also be an accidental misuse of the system or a more physical threat like fire or flood.

Trust Boundaries: Typically a trust boundary constitutes the components of the system under your direct control. All connections and data from systems outside of your direct control, including all clients and systems managed by other parties, should be consider untrusted and be validated at the boundary, before allowing further system interaction.

Vulnerability: A weakness that makes the system susceptible to attack or damage.