OWASP Secure Coding Practices Quick Reference Guide

Copyright and License

Table of Contents

Introduction

This document is technology agnostic and defines a set of general software security coding practices, in a checklist format, that can be integrated into the development lifecycle. Implementation of these practices will mitigate most common software vulnerabilities.

It is generally, much less expensive to build secure software than to correct security issues in a completed software package, 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 knowledge already 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 the design and implementation are 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:

- Implement a secure software development lifecycle
  - OWASP CLASP Project
- Clearly define roles and responsibilities
- Provide development teams with adequate software security training
- 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 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.

To protect the business from unacceptable risks associated with its reliance on software, it helps to understand what is meant by risk. 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. This means 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

Data Validation:

- Conduct all data validation on a trusted system (e.g., The server)
- Encode data to a common character set before validating (*Canonicalize*).
- 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 form fields, 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.
- Identify system *trust boundaries* and validate all data from external connections (e.g., Databases, file streams, etc.).
- Utilize a master encoding routine for inbound and outbound encoding.
- Validate all input against a "white" list of allowed characters.
- *Sanitize* any potentially hazardous characters that must be allowed, like: <, >, ", ', %, (,), &, +, \, \', \"
- Validate for expected data types.
- Validate data range.
- Validate data length.
- *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.
- 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)

Authentication and Password Management:

- Establish and utilize standard, tested, security services whenever possible
- 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.
- Validate the authentication data only on completion of all data input, especially for *sequential authentication* implementations.
- Error conditions 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.
- Use only POST requests to transmit authentication credentials.
- Only send passwords over an encrypted connection.
- Enforce password complexity requirements established by policy or regulation. An example might be 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 should only send an email to a pre-registered email address with a temporary link/password which lets the user reset the password.
- Temporary passwords and links should have a short expiration time.
- Password reset questions should support sufficiently random answers.
- Enforce the changing of temporary passwords on the next use.
- 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.
- Disable "remember me" functionality for password fields.
- Log all authentication failures.
- The last use (successful or unsuccessful) of a user account should be reported to the user at their next successful login.
- Segregate authentication logic and use redirection after login.
- If your application manages a credential store, it should ensure that only the one-way salted hashes of passwords are stored in the database, and that the table/file that stores the passwords and keys are write-able only by the application.
- Use a cryptographically strong one-way hash algorithm, such as SHA-256. Do not use the MD5 algorithm if it can be avoided.
- Implement monitoring to identify attacks against multiple user accounts, utilizing the same password. This attack pattern is used to bypass standard lockouts.
- Logout functionality should be available from all pages.
- Logout functionality should fully terminate the associated session or connection.

Authorization and Access Management:

- Use only server side session objects for making authorization decisions.
- Enforce authorization controls on every request, including server side scripts, "includes" and requests from rich client-side technologies like AJAX and Flash.
- Ensure that all directories, files or other resources outside the application's direct control have appropriate access controls in place.
- If *state data* must be stored on the client, use encryption and integrity checking on the server side to catch state tampering. The application should log all apparent tampering events.
- Enforce application logic flows to comply with business rules.
- Use a single site-wide component to check access authorization.
- Segregate privileged logic from other application code.
- 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.
- Create an Access Control Policy to document an application's business rules 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.
- If long authenticated sessions are allowed, periodically re-validate a user's authorization to ensure that their privileges have not changed.
- 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.).
- 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.

Session Management:

- Establish a session inactivity timeout that is as short as possible. It should be no more than several hours.
- If a session was established before login, close that session and establish a new session after a successful login.
- Do not allow concurrent logins with the same user ID.
- Use well vetted algorithms that ensure sufficiently random session identifiers.
- Session identifier creation must always be done on the server side.
- Do not pass session identifiers as GET parameters.
- Protect server side session data from unauthorized access by implementing appropriate access controls.
- Generate a new session identifier and deactivate the old one frequently.
- Generate a new session token if a user's privileges or role changes.
- Generate a new session token if the connection security changes from HTTP to HTTPS.
- Only utilize the system generated session identifiers for client side session management. Avoid using parameters or other client data for state management.
- Utilize per-session random tokens or parameters within web forms or URLs that are associated with sensitive server-side operations, like account management, to prevent *Cross Site Request Forgery* attacks.
- Utilize per-request random tokens or parameters to supplement the main session token for critical operations.
- Ensure cookies transmitted over an encrypted connection have the "secure" attribute set.
- Set cookies with the HttpOnly attribute, unless you specifically require client-side scripts within your application to read or set a cookie's value.
- The application or system should log attempts to connect with invalid or expired session tokens.
- 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.

Sensitive Information Storage or Transmission:

- Implement encryption for the transmission of all sensitive information.
- Encrypt highly sensitive stored information, like authentication verification data, even on the server side. Always use well vetted algorithms.
- 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.
- Do not store sensitive information in logs.
- Implement least privilege, restrict users to only the functionality, data and system information that is required to perform their tasks.
- Remove comments in production code.
- Remove unnecessary application and system documentation.
- Turn off verbose system messages, especially any associated with error conditions.
- The application should handle application errors and not rely on the server configuration.
- Do not include sensitive information in GET request parameters or at least filter that information from the HTTP referer, when linking to external sites.

System Configuration Management:

- Ensure servers, frameworks and system components are patched.
- Ensure servers, frameworks and system components are running the latest approved version.
- Use exception handlers that do not display debugging information.
- Disable unnecessary extended HTTP methods. If an extended HTTP method that supports file handling is required, utilize a well-vetted authentication mechanism such as WebDAV.
- If the webserver 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).
- Turn off directory listings.
- Ensure SSL certificates have the correct domain name, are not expired, and are installed with intermediate certificates if required.
- Restrict the web server, process and service accounts to the least privileges possible.
- Implement generic error messages and custom error pages that do not disclose system information.
- When exceptions occur, fail securely.
- Remove all unnecessary functionality and files.
- Remove any test code.
- Remove unnecessary information from HTTP response headers related to the OS, web-server version and application frameworks.
- Prevent disclosure of your directory structure and stop robots from indexing sensitive files and directories by moving them into an isolated parent directory and then "Disallow" that entire parent directory in the robots.txt file.
- Log all exceptions.
- Restrict access to logs.
- Log all administrative functions.
- Use a cryptographic hash function to validate log entry integrity.
- Implement asset management systems and register system components and software in them.

General Coding Practices:

- 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 tested and approved managed code rather than creating new unmanaged code for common tasks.
- Utilize locking to prevent multiple simultaneous requests to the application or use a synchronization mechanism to prevent race conditions.
- Explicitly initialize all your variables and other data stores, either during declaration or just before the first usage.
- Properly free allocated memory upon the completion of functions and at all exit points including error conditions.
- 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.

Database Security:

- Use strongly typed parameterized queries. Parameterized queries keep the query and data separate through the use of placeholders. The query structure is defined with place holders and then the application specifies the contents of each placeholder.
- Utilize input validation and if validation fails, do not run the database command.
- Ensure that variables are strongly typed.
- Escape meta characters in SQL statements.
- The application should use the lowest possible level of privilege when accessing the database.
- Use secure credentials for database access.
- Do not provide connection strings or credentials directly to the client. If this is unavoidable, encrypted them.
- Use stored procedures to abstract data access.
- Turn off any database functionality (e.g., unnecessary stored procedures or services).
- Eliminate default content.
- Disable any default accounts that are not required to support business requirements.
- Close the connection as soon as possible.
- 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.
- 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 web space. If this must be allowed, prevent or restrict the uploading of any file that can 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 hard coded 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, 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 hard coded paths.
- Never send the absolute file path to the client.
- Ensure application files and resources are read-only.
- Implement access controls for temporary files.
- Remove temporary files as soon as possible.

Memory Management:

- Utilize contextual specific validation..
- 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.).

Appendix A:

External References:

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

- Open Web Application Security Project (OWASP)
  http://www.owasp.org/index.php/Main_Page
  - OWASP CLASP Project
  - OWASP Development Guide Project
  - OWASP Enterprise Security API (ESAPI) Project
  - OWASP Application Security Verification Standard (ASVS) Project)
  - OWASP Legal Project
- 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

Security Advisory Sites:
These can be useful to check to ensure supporting infrastructure and frameworks do not have know vulnerabilities

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.

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

Authorization: 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.

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

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

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 included in the response to the client. (e.g. the body of an HTML document, an HTML attribute, within JavaScript, within a CSS or in a URL)

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.*)

Data Validation: Verification that the properties of all input and output data match what is expected by the application and that any potentially harmful data is made safe through the use of data removal, replacement, encoding, or escaping.

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.

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.

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 resulting from the occurrence of an undesired event; what would be the result of a vulnerability being exploited.

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

Memory Management: This is 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).

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.

Sensitive Information Storage or Transmission: A set of controls that help ensure the software handles the sending, receiving and storing of information in a secure manner.

Session Management: A set of controls that help ensure web applications handle http sessions 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.

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 Management: 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.