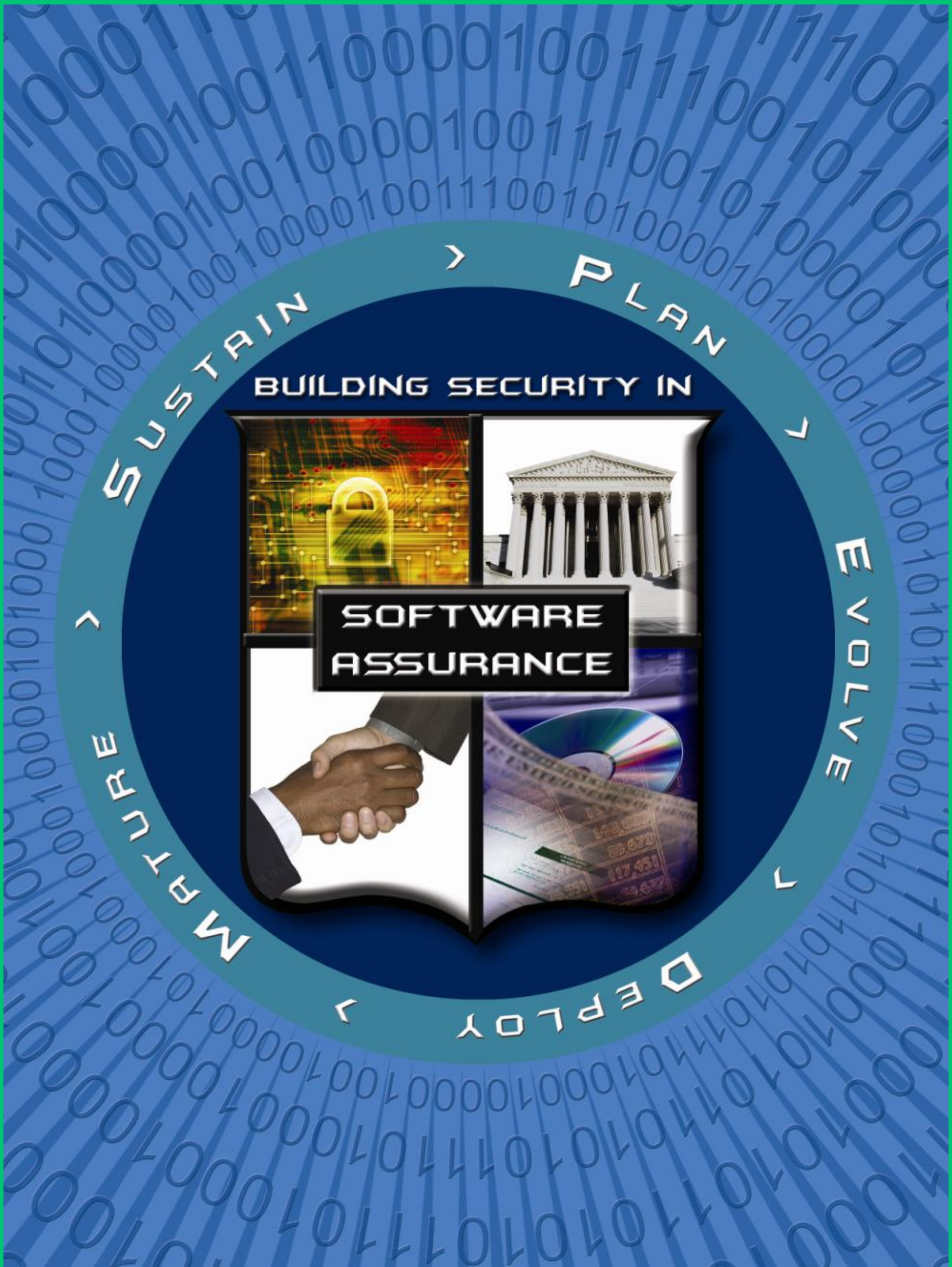
Architecture and Design Considerations for Secure Software

Software Assurance Pocket Guide Series:
Development, Volume V
Version 2.0, May 18, 2012

5



Software Assurance (SwA) Pocket Guide Resources

This is a resource for 'getting started' in selecting and adopting relevant practices for delivering secure software. As part of the Software Assurance (SwA) Pocket Guide series, this resource is offered as informative use only; it is not intended as directive or comprehensive. Rather it references and summarizes material in the source documents that provide detailed information. When referencing any part of this document, please provide proper attribution and reference the source documents, when applicable.

This volume of the SwA Pocket Guide series focuses on the practices and knowledge required to establish the architecture and high-level design for secure software during the Software Development Life Cycle (SDLC). It addresses design aspects such as threat modeling, misuse/abuse cases, and secure design patterns. The pocket guide covers design aspects of specific technologies such as mobile applications, databases, embedded systems, and web applications. It addresses formal methods and architectural design, principles for the design of secure software, and criteria for design review and verification. It describes key architecture and design practices for mitigating exploitable software weaknesses. Questions are offered for managers, in development and procurement, to aid in understanding whether the software development team has performed requisite practices to ensure the architecture and design sufficiently contributes toward the development of secure software.

At the back of this pocket guide are references, limitation statements, and a listing of topics addressed in the SwA Pocket Guide series. All SwA Pocket Guides and SwA-related documents are freely available for download via the SwA Community Resources and Information Clearinghouse at https://buildsecurityin.us-cert.gov/swa .

Acknowledgements

The SwA Forum and Working Groups function as a stakeholder mega-community that welcomes additional participation in advancing software security and refining SwA-related information resources that are offered free for public use. Input to all SwA resources is encouraged. Please contact Software.Assurance@dhs.gov for comments and inquiries. For the most up to date pocket guides, check the website at https://buildsecurityin.us-cert.gov/swa/.

The SwA Forum is a multi-disciplinary community composed of members of the government, industry, and academia. Meeting quarterly in SwA Forum and Working Groups, the community focuses on incorporating SwA considerations in acquisition and development processes relative to potential risk exposures that could be introduced by software and the software supply chain.

Participants in the SwA Forum's Processes & Practices Working Group collaborated with the Technology& Tools Working Group in developing the material used in this pocket guide with a goal of raising awareness on how to incorporate SwA throughout the Software Development Life Cycle (SDLC).

Information contained in this pocket guide comes primarily from the documents listed in the *Resource* boxes that appear throughout this pocket guide.

Special thanks to the Department of Homeland Security (DHS) National Cyber Security Division's Software Assurance team, Robert Seacord, and Dan Cornell; who provided much of the support to enable the successful completion of this guide and related SwA documents.

Resources

- "Software Security Assurance: A State-of-the-Art Report"(SOAR), Goertzel, Karen Mercedes, *et al.*, Information Assurance Technology Analysis Center (IATAC) of the DTIC. 31 July 2007. <http://iac.dtic.mil/iatac/download/security.pdf>.
- "Guide to the Software Engineering Body of Knowledge (SWEBOK)." IEEE Computer Society, 2004. <http://www2.computer.org/portal/web/swebok>.
- "Microsoft Security Development Lifecycle (SDL) – Process Guidance." Microsoft Developer Network (MSDN), <http://msdn.microsoft.com/en-us/library/84aed186-1d75-4366-8e61-8d258746bopq.aspx>.
- "The Ten Best Practices for Secure Software Development", Mano Paul, (ISC)^2, <https://www.isc2.org/uploadedFiles/(ISC)2_Public_Content/Certification_Programs/CSSLP/ISC2_WPIV.pdf>.

Overview

The Guide to the Software Engineering Body of Knowledge (SWEBOK) defines the design phase as both "the process of defining the architecture, components, interfaces, and other characteristics of a system or component" and "the result of [that] process." The software design phase is the software engineering life cycle activity where software requirements are analyzed in order to produce a description of the software's internal structure, which will be served as the basis for the software's implementation.

The software design phase consists of the architectural design and detailed design activities in the Software Development Life Cycle (SDLC). In the Waterfall model, these activities follow the software requirements analysis phase and precedes the implementation phase. However, the concepts presented in this pocket guide can be applied regardless of the development methodology employed. In any event, the practices and recommendations should be tailored to the realities of how the software is built.

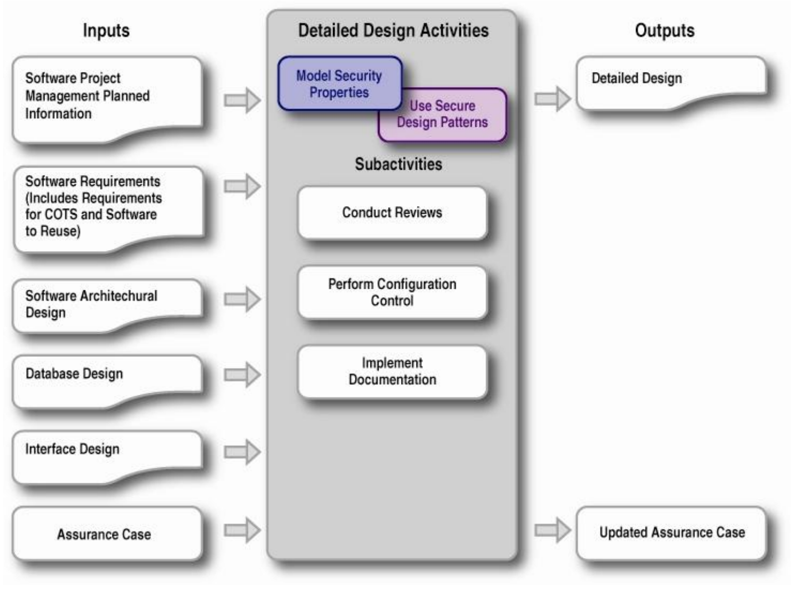
Table of Contents

Basic Concepts

Software architectural design, also known as top-level design, describes the software's structure, organization, and components. The architectural design allocates requirements to components identified in the design phase. Architecture describes components at an abstract level, leaving their implementation details unspecified. Some components may be modeled, prototyped, or elaborated at lower levels of abstraction. Top-level design activities include the design of interfaces among components in the architecture and can also include database design. Artifacts produced during the architectural design phase can include:

- Models, prototypes, simulations, and their related documentation;
- Preliminary user's manual;
- Preliminary test requirements;
- Documentation of feasibility; and
- Documentation of the traceability of requirements to the architecture design.

Figure 1 - Architecture Design With Assurance Activities


Inputs
Detailed Design Activities
Outputs
Software Project Management Planned Information
Model Security Properties
Use Secure Design Patterns
Detailed Design
Software Requirements (Includes Requirements for COTS and Software to Reuse)
Subactivities
Conduct Reviews
Software Architechtural Design
Perform Configuration Control
Database Design
Implement Documentation
Interface Design
Assurance Case
Updated Assurance Case

Software detailed design consists of describing each component sufficiently to allow for its implementation. Detailed design activities define data structures, algorithms, and control information for each component in a software system. The State-of-the-Art Report (SOAR) Figure 1 (modified for this guide) illustrates the architectural and detailed design phases as they would be implemented for a standard software life cycle depicted in IEEE Standard 1074-2006 with security assurance activities and artifacts included. Figure 1 can be modified to represent non-Waterfall software life cycles accordingly.

Each input depicted should include specific attention to security goals. To decrease the number of design vulnerabilities, special attention should be devoted to security issues captured during threat modeling, requirements analyses, and early architecture phases. In general, a design vulnerability is a flaw in a software system's architecture, specification, or high-level or low-level design that results from a fundamental mistake or oversight in the design phase. These types of flaws often occur because of incorrect assumptions made about the run-time environment or risk exposure that the system will encounter during deployment.

In his article "Lessons Learned from Five Years of Building More Secure Software," Michael Howard makes the point that many software security vulnerabilities are not coding issues but design issues. When one is exclusively focused on finding security issues in code, one risks missing out on entire classes of vulnerabilities. Some security issues, not syntactic or code related (such as business logic flaws), cannot be detected in code and need to be identified by performing threat models and abuse case modeling during the design stage of the SDLC.

The best time to influence a project's security design is early in its life cycle. Functional specifications may need to describe security features or privacy features that are directly exposed to users, such as requiring user authentication to access specific data or user consent before use of a high privacy-risk feature. Design specifications should describe how to implement these features and how to implement all functionality as secure features. Secure features are defined as features with functionality that are well engineered with respect to security, such as rigorously validating all data before processing it or using of robust cryptographic APIs. It is important to consider security issues carefully and early when you design features and to avoid attempts to add security and privacy near the end of a project's development.

Enterprise architecture (EA) and enterprise security is not directly addressed in this pocket guide. However, the perspective is valuable in that it permits organizations to invest in security solutions that protect the entire enterprise, while allocating costs and controls where they are most needed as determined by the enterprise. For example, the Federal Enterprise Architecture Security and Privacy Profile provides a publicly accessible perspective on the role of security across the enterprise. Security and privacy architecture reference models promote enterprise-wide interoperability and help standardize and consolidate security and privacy capabilities. Layering security and privacy over organization performance objectives, business processes, service-components, technologies, and data helps ensure that each aspect of the business receives appropriate security and privacy attention. Establishing a common methodology requires "the coordinated efforts of business leaders and functional domain experts, including security, privacy, enterprise architecture, and capital planning." EA encourages the incorporation of diverse stakeholders, "such as representatives of the acquisitions, contracts, and legal departments." Inclusion of security in the process of business transformation will promote effective and economical security solutions that are appropriate to the risk appetite of the business units.

Resources

» "Software Security Assurance: A State-of-the-Art Report" (SOAR). Goertzel, Karen Mercedes, *et al.,* Information Assurance Technology Analysis Center (IATAC) of the DTIC, 31 July 2007. <http://iac.dtic.mil/iatac/download/security.pdf>.

» "IEEE Standard for Developing a Software Project Life Cycle Process." IEEE Computer Society, <http://standards.ieee.org/findstds/standard/1074-2006.html>.

» "Lessons Learned from Five Years of Building More Secure Software", Michael Howard, Microsoft Developer Network (MSDN), November 2007. <http://msdn.microsoft.com/en-us/magazine/cc163310.aspx>.

» "Federal Enterprise Architecture Security and Privacy Profile." Federal Enterprise Architecture of the United States Government (FEA), September 2010. <http://www.cio.gov/Documents/FEA-Security-Privacy-Profile-v3-09-30-2010.pdf>.

Modeling From the Attacker's Perspective

Unified Markup Language (UML), developed by the Object Management Group (OMG), is a widely-used specification for modeling software. UML provides the ability to describe the software architecture using various types of diagrams. UML diagrams describe application states, information flow, components interaction, and more. UML is quite complex, so explaining it in depth is beyond the scope of this pocket guide. Nevertheless, brief descriptions of some of the diagrams available in UML are provided. For additional information, visit the UML resource page or consult one of the multiple books available on the subject.

Use case diagrams describe an application in action. The emphasis is on what a system does rather than how. Use cases can be represented either in text or graphics, and there is no restriction on what the use case diagrams should include or look like.

A Class diagram gives an overview of a system by showing its classes (i.e., basic concepts which each comprise a defined state and attendant behavior) and the relationships among them. It lays out how an application is modeled, how classes interact with each other, and the relationships between modules for an Object-Oriented design.

A Component diagram describes the relationship of system components (software modules) and depicts the component interfaces. Examples of components include data bases, web applications, etc.

Other modeling diagrams include:

» Object diagrams,

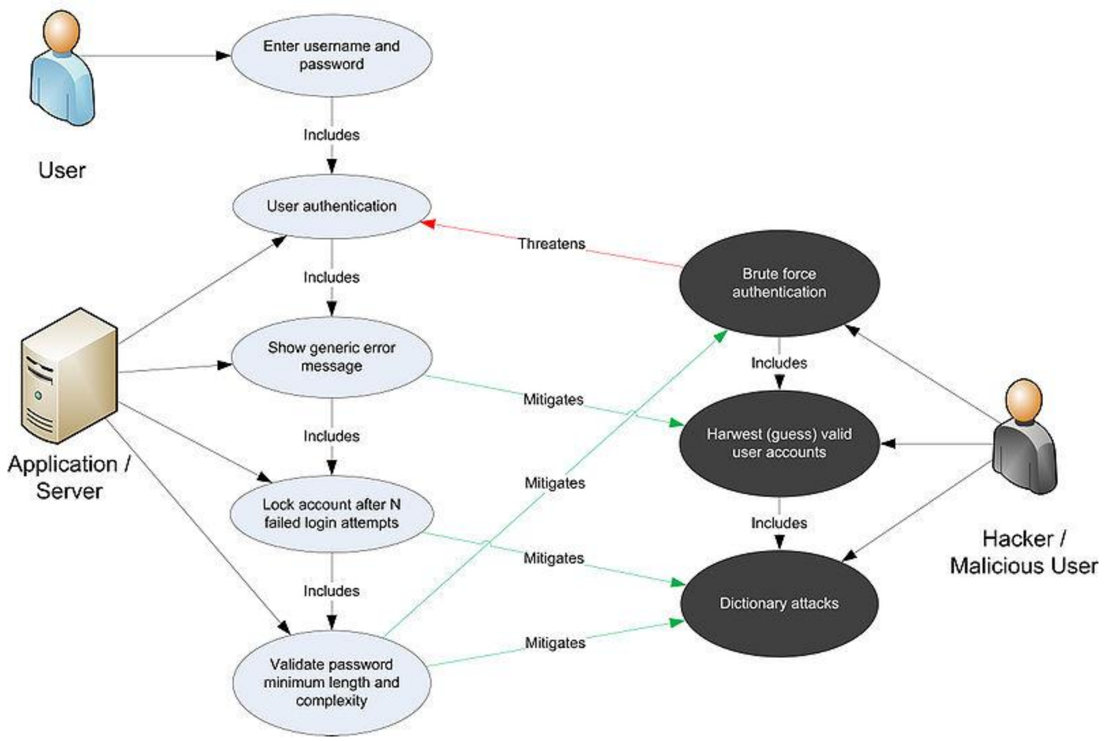
» Sequence diagrams,

» Collaboration diagrams,
» Statechart diagrams,
» Activity diagrams, and
» Deployment diagrams.

Some people in the SwA community feel that UML does not allow for the capture of security properties and does not include a model for use or abuse cases. Successful techniques for this purpose include threat modeling, data flow diagrams, abuse cases, and attack trees.

Misuse/Abuse Cases – Misuse cases are similar to UML use cases, except that they are meant to detail common attempted abuses of the system. Like use cases, misuse cases require understanding the services that are present in the system. A use case generally describes behavior that the system owner wants the system to implement. Misuse cases apply the concept of a negative scenario—that is, a situation that the system's owner does *not* want to occur. For an in-depth view of misuse cases, see Gary McGraw's "Misuse and Abuse Cases: Getting Past the Positive" at the Build Security In (BSI) portal at https://buildsecurityin.us-cert.gov/; a direct link is available under resources at the end of this section.

Figure 2 - Misuse Case example (source: OWASP's Testing Guide)


User
Enter username and password
Includes
User authentication
Threatens
Includes
Show generic error message
Includes
Lock account after N failed login attempts
Includes
Validate password minimum length and complexity
Application / Server
Mitigates
Mitigates
Mitigates
Mitigates
Brute force authentication
Includes
Harwest (guess) valid user accounts
Includes
Dictionary attacks
Hacker / Malicious User

Misuse cases help organizations see their software in the same light attackers do. Just as use-case models have proven quite helpful for the functional specification of requirements, misuse cases and use cases can improve the efficacy of eliciting security requirements. Guttorm Sindre and Andreas Opdahl extended use-case diagrams with misuse cases to represent the actions that systems should prevent in tandem with those that they should support for security and privacy requirements. There are several templates for misuse and abuse cases provided by Sindre and Opdahl. Figure 2 shows an example of a use/misuse case diagram from OWASP's Testing Guide. The use case diagram demonstrates the actions that both the user and the application perform in the particular scenario. The misuse case diagram demonstrates the actions that can be taken by an attacker to hack into the system in the particular scenario. The two are linked together by arrows showing which of the attacker's actions threaten the actions of the user/application as well as which of the user/application's actions thwart the actions of the attacker. Making a diagram like this can point out possible security holes in the system.

Use cases describe system behavior in terms of functional (end-user) requirements. Misuse cases and use cases may be developed from system to subsystem levels—and lower as necessary. Lower level cases may draw attention to underlying problems not considered at higher levels and may compel system engineers to reanalyze the system design. Misuse cases are not a top-down method, but they provide opportunities to investigate and validate the security requirements necessary to accomplish the system's mission.

As with normal use cases, misuse cases require adjustment over time. Particularly, it is common to start with high-level misuse cases and then refine them as the details of the system are better understood. Determining misuse cases generally involves informed brainstorming activity among a team of security and reliability experts. In practice, a team of experts asks questions of a system's designers to help identify the places where the system is likely to have weaknesses by assuming the role of an attacker and thinking like an attacker. Such brainstorming involves a careful look at all user interfaces (including environmental factors) and considers events that developers assume a person can't or won't do. There are three good starting points for structured brainstorming:

» First, one can start with a pre-existing knowledge base of common security problems and determine whether an attacker may have cause to think such a vulnerability is possible in the system. Then, one should attempt to describe how the attacker will leverage the problem.

» Second, one can brainstorm on the basis of a list of system resources. For each resource, attempt to construct misuse cases in connection with each of the basic security services: authentication, confidentiality, access control, integrity, and availability.

» Third, one can brainstorm on the basis of a set of existing use cases. This is a far less structured way to identify risks in a system, yet it is good for identifying representative risks and for ensuring the first two approaches did not overlook any obvious threats. Misuse cases derived in this fashion are often written in terms of a valid use and then annotated to have malicious steps.

The OWASP Comprehensive, Lightweight application Security Process (CLASP) process recommends describing misuse cases as follows:

» A system will have a number of predefined roles and a set of attackers that might reasonably target instances of the system under development. Together these should constitute the set of actors that should be considered in misuse cases.

» As with traditional use cases, establish which actors interact with a use case — and how they do so — by showing a communicates-association. Also as traditionally done, one can divide use cases or actors into packages if they become too unwieldy.

» Important misuse cases should be represented visually, in typical use case format, with steps in a misuse set off (e.g., a shaded background). This should particularly be done when the misuse is effectively an annotation of a legitimate use case.

» Those misuse cases that are not depicted visually but are still important to communicate to the user should be documented, as should any issues not handled by the use case model.

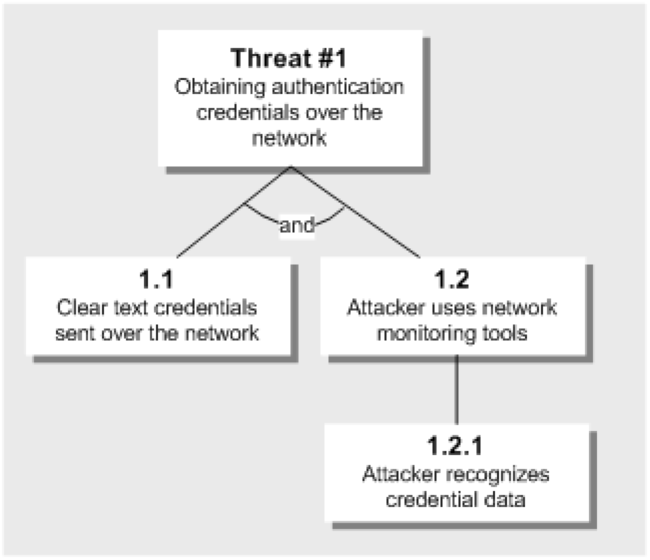
Attack Trees – Attack trees provide a formal, methodical technique for describing the security of systems, based on varying attacks. Attacks against a system are represented in a tree structure, with the goal as the root node and the different routes of achieving that goal as the leaf nodes. Figure 3 is an example, provided by Microsoft, where the main goal of obtaining the "authentication credentials" is represented as the root node and the attacker's different techniques to "obtain the credentials" are represented as the leaf nodes.

To add more substance to the attack tree, assigned values can be added to the leaf nodes. Some optional values to assign to the leaf nodes are:

» Adding the 'AND' nodes which requiring two leaf nodes to be completed before going to the parent node. Every node that isn't an 'AND' node is considered an 'OR' node.

- Adding monetary values, to the leaf nodes, to represent the amount of money required for an attacker to accomplish the attack. With assigned monetary values, the designer can create countermeasures that make the cheaper attack routes expensive or prevent the attacker from taking a certain route altogether.

Figure 3 – Attack Tree Example (Microsoft)


Threat #1
Obtaining authentication credentials over the network
and
1.1
Clear text credentials sent over the network
1.2
Attacker uses network monitoring tools
1.2.1
Attacker recognizes credential data

Threat Modeling – A **threat** is a potential occurrence, malicious or otherwise, that might damage or compromise system resources. Threat modeling is a systematic process that is used to identify threats and vulnerabilities in software and has become popular technique to help system designers think about the security threats that their system might face. Therefore, threat modeling can be seen as risk assessment for software development. It enables the designer to develop mitigation strategies for potential vulnerabilities and helps them focus their limited resources and attention on the parts of the system most "at risk." It is recommended that all software systems have a threat model developed and documented. Threat models should be created as early as possible in the SDLC and should be revisited as the system evolves and development progresses. The National Institute of Standards and Technology (NIST) 800-30 standard for risk assessment can be used as a guideline in developing a threat model. This approach involves:

- **Decomposing the application** – understand, through a process of manual inspection, how the application works, its assets, functionality, and connectivity.
- **Defining and classifying the assets** – classify the assets into tangible and intangible assets and rank them according to business importance.
- **Exploring potential vulnerabilities** – whether technical, operational, or managerial.
- **Exploring potential threats** – develop a realistic view of potential attack vectors from an attacker's perspective, by using threat scenarios or attack trees.
- **Creating mitigation strategies** – develop mitigating controls for each of the threats deemed to be realistic. The output from a threat model itself can vary but is typically a collection of lists and diagrams. The OWASP Code Review Guide at http://www.owasp.org/index.php/Application_Threat_Modeling outlines a methodology that can be used as a reference for the testing for potential security flaws.

Resources

- “OMG Unified Modeling Language: UML® Resource Page.” Object Management Group [OMG], 30 November 2010. <http://www.uml.org/>.
- “Misuse and Abuse Cases: Getting Past the Positive.” Paco Hope, Gray McGraw, and Annie Anton, IEEE Security & Privacy, May 2004. <https://buildsecurityin.us-cert.gov/bsi/resources/414-BSI/125-BSI.html>.
- OWASP CLASP Project. “OWASP Comprehensive, Lightweight Application Security Process (CLASP) project.” The Open Web Application Security Project [OWASP], 31 March 2006. <http://www.owasp.org/index.php/Category:OWASP_CLASP_Project>.
- “OWASP Testing Guide: 2008 version 3.0.” OWASP, 30 November 2010. <http://www.owasp.org/images/5/56/OWASP_Testing_Guide_v3.pdf>.
- “Eliciting Security Requirements by Misuse Cases.” Andreas Opdahl and Guttorm Sindre, 06 August 2002. <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=891363>.
- “Attack Trees.” Bruce Schneier, December 1999. <http://www.schneier.com/paper-attacktrees-ddj-ft.html>
- “How To: Create a Threat Model for a Web Application at Design Time.” J.D. Meier, Alex Mackman, Blaine Wastell, Microsoft , May 2005. < http://msdn.microsoft.com/en-us/library/ff647894.aspx>
- “Risk Management Guide for Information Technology Systems.” NIST 800-30. July 2002. <http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf>.
- “Application Threat Modeling.” OWASP. 7 January 2011. <https://www.owasp.org/index.php/Application_Threat_Modeling>.

Design Principles for Secure Software

Developers need to know secure software design principles and how they are employed in the design of resilient and trustworthy systems. Two essential concepts of design include abstraction and decomposition:

Abstraction is a process for reducing the complexity of a system by removing unnecessary details and isolating the most important elements to make the design more manageable.

Decomposition (also known as factoring) is the process of describing the generalizations that compose an abstraction. One method, top-down decomposition, involves breaking down a large system into smaller parts. For object-oriented designs, the progression would be application, module, class, and method.

Other secure software design principles are detailed in a multitude of books, white papers, web portals, and articles. This section will highlight two approaches and provide reference resources for additional research. The first one is derived from the developers guide “Enhancing the Development Life Cycle to Produce Secure Software” that describes three general principles summarized on Table 1:

Table 1- Adapted from "Enhancing the Development Life Cycle to Produce Secure Software"

General Principle	Key Practices	Benefits	Examples and Practices
Minimize the number of high-consequence targets	Principle of least privilege	Minimizes the number of actors in the system that are granted high levels of privilege, and the amount of time any actor can hold onto its privileges.	In a traditional Web portal application, the end user is only allowed to read, post content, and enter data into HTML forms, while the Webmaster has all the permissions.
	Separation of privileges, duties, and roles	Ensures that no single entity (human or software) should have all the privileges required to modify, delete, or destroy the system, components and resources.	Developers should have access to the development and test code/systems; however, they should not have access to the production system. If developers have access to the production system, they could make unauthorized edits that could lead to a broken application or add malicious code for their personal gain. The code needs to go through the appropriate approvals and testing before being deployed into production. On the other hand, administrators should be able to deploy the package into production but should not have the ability to edit the code.
	Separation of domains	Separation of domains makes separation of roles and privileges easier to implement.	Database administrators should not have control over business logic and the application administrator should not have control over the database.
Don't expose vulnerable or high-consequence components	Keep program data, executables, and configuration data separated	Reduces the likelihood that an attacker who gains access to program data will easily locate and gain access to program executables or control/configuration data.	On Unix or Linux systems, the *chroot* "jail" feature of the standard operating system access controls can be configured to create an isolated execution area for software, thus serving the same purpose as a Java or Perl "sandbox."
	Segregate trusted entities from untrusted entities	Reduces the exposure of the software's high-consequence functions from its high-risk functions, which can be susceptible to attacks.	Java and Perl's sandboxing and .NET's Code Access Security mechanism in its Common Language Runtime (CLR) assigns a level privilege to executable(s) contained within it. This privilege level should be the minimal needed by the function(s) to perform its normal expected operation. If any anomalies occur, the sandbox/CLR will generate an exception and an exception handler will prevent the executable from performing the unexpected operation.

Table 1- Adapted from "Enhancing the Development Life Cycle to Produce Secure Software"

General Principle	Key Practices	Benefits	Examples and Practices
	Minimize the number of entry and exit points	Reduces the attack surface.	Firewalls provide a single point of contact (called a chokepoint) that allows the administrator control of traffic coming into or out of the network. Like a firewall, strive for one entry point into any software entity (function, process, module component) and ideally one exit point.
	Assume environment data is not trustworthy	Reduces the exposure of the software to potentially malicious execution environment components or attacker-intercepted and modified environment data.	Java Platform, Enterprise Edition (Java EE) components run within "contexts" (e.g. System Context, Login Context, Session Context, Naming and Directory Context, etc.) that can be relied on to provide trustworthy environment data at runtime to Java programs.
	Use only trusted interfaces to environment resources	This practice reduces the exposure of the data passed between the software and its environment.	Application-level programs should call only other application-layer programs, middleware, or explicit APIs to system resources. Applications should not use APIs intended for human users rather than software nor rely on a system-level tool (versus an application-level tool) to filter/modify the output.
Deny attackers the means to compromise	Simplify the design	This practice minimizes the number of attacker-exploitable vulnerabilities and weaknesses in the system.	The software should limit the number of states, favor deterministic processes over non-deterministic processes, use single-tasking rather than multitasking whenever practical, use polling rather than interrupts, etc.
	Hold ***all*** actors accountable	This practice ensures that all attacker actions are observed and recorded, contributing to the ability to recognize and isolate/block the source of attack patterns.	Enforcing accountability with the combination of auditing and non-repudiation measures. Auditing amounts to security-focused event logging to record all security-relevant actions performed by the actor while interacting with the system. Audits are after-the-fact and often can be labor intensive; Security-Enhanced Linux (SELinux) can be used to enforce data access using security labels. Non-repudiation measures, most often a digital signature, bind proof of the identity of the actor responsible for modifying the data.

Table 1- Adapted from "Enhancing the Development Life Cycle to Produce Secure Software"			
General Principle	**Key Practices**	**Benefits**	**Examples and Practices**
	Timing, synchronization, and sequencing should be simplified to avoid issues.	Modeling and documenting timing, synchronization, and sequencing issues will reduce the likelihood of race conditions, order dependencies, synchronization problems, and deadlocks.	Whenever possible make all individual transactions atomic, use multiphase commits for data "writes", use hierarchical locking to prevent simultaneous execution of processes, and reduce time pressures on system processing.
	Make secure states easy to enter and vulnerable states difficult to enter	This practice reduces the likelihood that the software will be allowed to inadvertently enter a vulnerable state.	Software should always begin and end its execution in a secure state.
	Design for controllability	This practice makes it easier to detect attack paths, and disengage the software from its interactions with attackers. Caution should be taken when using this approach since it can open a whole range of new attack vectors.	To increase the software controllability, design the software to have the ability to self-monitor and limit resources usage, provide exception handling, error handling, anomaly handling, and provide feedback that enables all assumptions and models to be validated before decisions are taken.
	Design for secure failure	Reduces the likelihood that a failure in the software will leave it vulnerable to attack.	Implement watchdog timers to check for the "I'm alive" signals from processes and use exception handling logic to correct actions before a failure can occur.

The second set of secure software design principles is from the highly-regarded paper "***The Protection of Information in Computer Systems***" by Saltzer and Schroeder, as shown on Table 2.

Table 2- Adapted from Saltzer & Shroeder "The Protection of Information in Computer Systems"	
Design Principle	**What it Means**
Economy of mechanism	Keep the design as simple and small as possible.
Fail-safe defaults	Base access decisions on permission rather than exclusion. This principle means that the default is lack of access, and the protection scheme identifies conditions under which access is permitted.
Complete mediation	Every access to every object must be checked for authority. This principle, when systematically applied, is the primary underpinning of the protection system. It forces a system-wide view of access control, which in addition to normal operation includes initialization, recovery, shutdown, and maintenance.
Open design	The design should not be secret. The mechanisms should not depend on the ignorance of potential attackers, but rather on the possession of specific keys or passwords that are easily protected.
Separation of privilege	A protection mechanism that requires two keys to unlock is more robust and flexible than one that allows access to the presenter with only a single key. Two keys apply to any situation in which two or more conditions must be met before access is granted.
Least privilege	Every program and every user of the system should operate using the least set of privileges necessary to complete the job.

Table 2- Adapted from Saltzer & Shroeder "The Protection of Information in Computer Systems"

Design Principle	What it Means
Least common mechanism	Minimize the amount of mechanism common to more than one user and depended on by all users. Every shared mechanism represents a potential information path between users and must be designed with great care to be sure it does not unintentionally compromise security.
Psychological acceptability	It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.

Resources

» "Enhancing the Development Life Cycle to Produce Secure Software (EDLC)." Goertzel, Karen Mercedes, DHS SwA Forum Process and Practices Working Group, October 2008. <https://www.thedacs.com/techs/enhanced_life_cycles/>.

» "Software Assurance: A Curriculum Guide to the Common Body of Knowledge to Produce, Acquire and Sustain Secure Software." Software Assurance Workforce Education and Training Working Group, DHS Build Security In (BSI) portal, October 2007. <https://buildsecurityin.us-cert.gov/daisy/bsi/940-BSI/version/default/part/AttachmentData/data/CurriculumGuideToTheCBK.pdf>.

» "The Protection of Information in Computer Systems." Saltzer and Schroeder, <http://web.mit.edu/Saltzer/www/publications/protection/>.

» "The Ten Best Practices for Secure Software Development." Paul, Mano, (ISC)^2, <https://www.isc2.org/uploadedFiles/(ISC)2_Public_Content/Certification_Programs/CSSLP/ISC2_WPIV.pdf>.

» "Domain Separation by Construction." William Harrison, Mark Tullsen, and James Hook, <http://citeseer.ist.psu.edu/viewdoc/download;jsessionid=22DD6A3A0797FDD2E6836D9DD86B1ED8?doi=10.1.1.86.9631&rep=rep1&type=pdf>.

Secure Design Patterns

A software design pattern is a general repeatable solution to a recurring software engineering problem. Secure design patterns are descriptions or templates describing a general solution to a security problem that can be applied in many different situations. They provide general design guidance to eliminate the introduction of vulnerabilities into code or mitigate the consequences of vulnerabilities. Secure design patterns are not restricted to object-oriented design approaches but may also be applied to procedural languages. Secure design patterns provide a higher level of abstraction than secure coding guidelines. Secure design patterns differ from security patterns in that they do not describe specific security mechanisms such as access control, authentication, authorization and logging. Whereas security patterns are necessarily restricted to security-related functionality, secure design patterns can (and should) be used across all functional areas of a system.

The ***Secure Design Patterns*** technical report, by the Software Engineering Institute, categorizes three general classes of secure patterns according to their level of abstraction: architecture, design, and implementation. This section provides a brief summary of the architecture and design patterns. The technical report includes sample code that implements these patterns.

Architectural-level Patterns

Architectural-level patterns focus on the high-level allocation of responsibilities between different components of the system and define the interaction between those high-level components and include:

Distrustful Decomposition – The intent of the Distrustful Decomposition secure design pattern is to move separate functions into mutually untrusting programs, thereby reducing the attack surface of the individual programs that make up the system's functionality and the data exposed to an attacker if one of the mutually untrusting programs is compromised. This pattern applies to systems where files or user-supplied data must be handled in a number of different ways by programs running with varying privileges and responsibilities.

Privilege Separation (PrivSep) – The intent of the PrivSep pattern is to reduce the amount of code that runs with special privileges; without affecting or limiting the functionality of the program. The PrivSep pattern is a more specific instance of the Distrustful Decomposition pattern. In general, the PrivSep pattern is applicable if the system performs a set of functions that:

- Do not require elevated privileges;
- Have relatively large attack surfaces;
- Have significant communication with untrusted sources; and
- Make use of complex, potentially error-prone algorithms.

Defer to Restricted Application or Area – The intent of this pattern is to clearly separate "functionality that requires elevated privileges" from "functionality that does not require elevated privileges." The "defer to restricted application or area" pattern is applicable to systems:

- That run by users who do not have elevated privileges;
- Where some (possibly all) of the functionality of the system requires elevated privileges; or
- Where the system must verify that the current user is authorized to execute any functionality that requires elevated privileges.

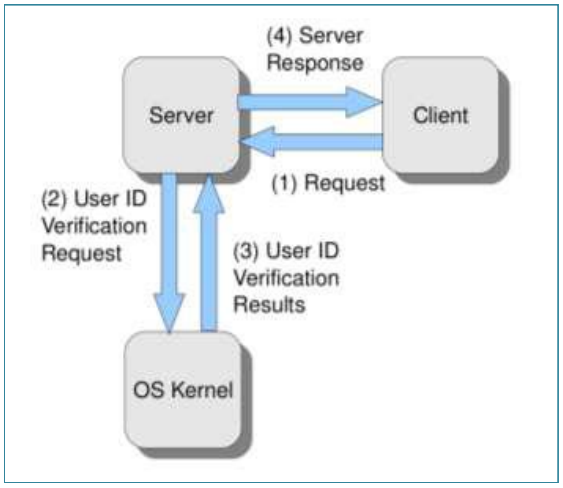

(4) Server Response
Server
Client
(1) Request
(2) User ID Verification Request
(3) User ID Verification Results
OS Kernel

Figure 4: General Structure of Defer to Kernel Pattern (source: Secure Design Patterns)

One way this can be done is by taking advantage of existing user verification functionality available at the kernel level. However this may not always be the best course of action, especially in the case of web applications in which untrusted data could come over the Internet and should not be sent to the kernel level. An alternative to this is to use a central application or database that has restricted access and can perform the same user verification actions as the kernel. Figure 4 depicts the general structure of the "defer to restricted application or area" pattern, using the kernel as its reference.

The three architectural-level patterns described in this section provide techniques to prevent improper authorization and promote privilege control. The Common Weakness Enumeration (CWE) [see http://cwe.mitre.org] defines a unified, measurable set of software weaknesses such as improper authorization and execution with unnecessary privileges. Common techniques for privilege control and proper authorizations provided by CWE are:

- Divide applications into anonymous, normal, privileged, and administrative areas (CWE-285).
- Reduce the attack surface by mapping roles with data and functionality and use role-based access control (RBAC) to enforce the roles at the appropriate boundaries (CWE-285).
- Use authorization frameworks such as the JAAS Authorization Framework and the OWASP ESAPI Access Control feature (CWE-285).
- For web applications, make sure that the access control mechanism is enforced correctly, at the server-side, on every page. Users should not be able to access any unauthorized functionality or information by simply requesting direct

access to that page. (One way to do this is to ensure that all pages containing sensitive information are not cached and to restrict access to requests that are accompanied by an active and authenticated session token.) (CWE-285)

- Perform access control checks related to the business logic (CWE-285).
- Wrap and centralize any functionality that requires additional privileges, such as access to privileged operating system resources, and isolate the privileged code as much as possible from other code. Raise the privileges as late as possible, and drop them as soon as possible. Protect all possible communication channels that could interact with privileged code, such as a secondary socket that is only intended for access by administrators (CWE-250).

For additional information on privilege control, see the mitigations for CWE 285 "Improper Access Control (Authorization)" in the Key Practices for Mitigating the Most Egregious Exploitable Software Weaknesses pocket guide.

Design-level Patterns

Design-level patterns describe how to design and implement pieces of a high-level system component. These patterns address problems in the internal design of a single high-level component. Design level patterns do not address the interaction of high-level components between themselves. The six design-level patterns defined in the ***Secure Design Patterns*** technical report are:

Secure Factory – The intent of the Secure Factory design pattern is to separate the security dependent logic, involved in creating or selecting an object, from the basic functionality of the created or selected object. The Secure Factory secure design pattern is only applicable if:

- The system constructs different versions of an object based on the security credentials of the user/operating environment.
- The available security credentials contain all of the information needed to select and construct the correct object. No other security-related information is needed.

If the requirements are met, the Secure Factory secure design pattern operates as follows:

1. A caller asks an implementation of the Secure Factory pattern for the appropriate object given a specific set of security credentials
2. The Secure Factory pattern implementation uses the given security credentials to select and return the appropriate object.

Specializations of the Secure Factory secure design pattern are the Secure Strategy Factory and the Secure Builder Factory, as described in the next sections of this pocket guide.

Secure Strategy Factory – As an extension of the Secure Factory secure design pattern, the Secure Strategy Factory pattern provides an easy to use and modifiable method for selecting the appropriate strategy object (an object implementing the Strategy pattern). The Strategy pattern provides a means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable during runtime. The strategy object is then used to perform a task based on the security credentials of a user or environment. The Secure Strategy Factory pattern operates as follows:

1. A caller asks an implementation of the Secure Strategy Factory pattern for the appropriate strategy to perform a general system function given a specific set of security credentials.
2. The Secure Strategy Factory pattern implementation uses the given security credentials to select and return the appropriate object implementing the Strategy pattern that will correctly perform the desired general system function.

Secure Builder Factory – The intent of the Secure Builder Factory design pattern is to separate the security dependent rules, involved in creating a complex object, from the basic steps involved in the actual creation of the object. A *complex object* is a library object made up of many interrelated elements or digital objects. The Secure Builder Factory pattern operates as follows:

1. A caller asks an implementation of the Secure Builder Factory pattern for the appropriate builder to build a complex object given a specific set of security credentials.

2. The Secure Builder Factory pattern implementation uses the given security credentials to select and return the appropriate object implementing the Builder pattern. The Builder pattern's intent is to abstract the steps of constructing an object so that difference implementations of these steps can construct difference representations of objects. The selected object will then be used to correctly build a complex object given the security rules identified by the given security credentials.

Secure Supply Chain of Responsibility – The intent of this pattern is to decouple the logic that determines user/environment-trust dependent functionality from the portion of the application requesting the functionality, simplify the logic that determines user/environment-trust dependent functionality, and makes it relatively easy to dynamically change the user/environment-trusted dependent functionality.

Secure State Machine – The intent of the Secure State Machine pattern is to allow a clear separation between security mechanisms and user-level functionality. This is achieved by implementing the security and user-level functionality as two separate state machines. This pattern is applicable if:

» The user-level functionality can be cleanly represented as a finite state machine, or

» The access control model for the state transition operations in the user-level functionality state machine can also be represented as a state machine.

When developing state machines, CWE-642 "External Control of Critical State Data" provides the following mitigation techniques:

» Use a framework that maintains the state automatically with a stateless protocol, such as HTTP. Examples include ASP.NET View State and the OWASP ESAPI Session Management feature. Be careful of language features that provide state support, since these might be provided as a convenience to the programmer and may not be considering security.

» Don't keep state information on the client without using encryption or integrity checking, or otherwise having a mechanism on the server-side to catch state tampering. When storing state information on the client use a message authentication code (MAC) algorithm, such as Hash Message Authentication Code (HMAC), provide an algorithm with a strong hash function, and salt the hash. Provide mutable classes with a clone method.

» Don't allow any application user to affect state directly in any way other than through legitimate actions leading to state transitions.

Additional information on CWE-642 "External Control of Critical State Data" can be found in the Key Practices for Mitigating the Most Egregious Exploitable Software Weaknesses pocket guide.

Secure Visitor – Secure systems may need to perform various operations on hierarchically structured data where each node in the data hierarchy may have different access restrictions; that is, access to data in different nodes may be dependent on the role/credentials of the user accessing the data. The Secure Visitor pattern allows nodes to lock themselves against being read by a visitor unless the visitor supplies the proper credentials to unlock the node. Once the Secure Visitor is defined, the only way to access a locked node is with a Visitor, this helps prevent unauthorized access to nodes in the data structure. A Visitor is a class with methods that require unlocked node objects. The Secure Visitor pattern is applicable if, among other things, the system nodes in the hierarchical data have different access privileges. For more information on Secure Visitor, please visit the Secure Design Patterns technical report at www.cert.org/archive/pdf/09tr010.pdf.

Resources

» "Secure Design Patterns." Chad Dougherty, Kirk Sayer, Robert Seacord, David Svoboda, and Kazuya Togashi, Software Engineering Institute, October 2009. <www.cert.org/archive/pdf/09tr010.pdf>.

» "Software Security Assurance: A State-of-the-Art Report" (SOAR). Goertzel, Karen Mercedes, *et al.,* Information Assurance Technology Analysis Center (IATAC) of the DTIC, 31 July 2007. <http://iac.dtic.mil/iatac/download/security.pdf>.

Secure Architecture and Design of Web Applications

As websites evolve from static to dynamic, the web application's attack surface will increase. Traditional network firewalls are ineffective in detecting and blocking application layer attacks. According to Barracuda Networks, web application attacks account for only 54 percent of all data breaches, but they account for 92 percent of stolen records. In order to properly protect the server and the users, the architecture and design of the web application must be properly implemented.

Securing the Web Application

In the course of architecting and designing a web application, OWASP recommends analyzing the security in the application's architecture, application's design, and network security (e.g. firewalls, remote access). In addition to following standard software architecture and design guidelines, web applications should also include proper authentication, authorization, cookie management, cryptographic techniques, input validation, error handling, logging, and cryptography. Table Table 3 provides examples and threat mitigation techniques, for web applications, based on the OWASP's Code Review Guide:

Table 3 – OWASP's Code Review Guide Recommendations

General Principle	Recommendations
Authentication	» Ensure all internal and external connections (user and entity) go through an appropriate and adequate form of authentication. » Ensure all pages enforce the requirement for authentication. » Pass authentication credentials or sensitive information only via HTTP "POST" method, do not accept HTTP "GET" method. » Ensure authentication credentials do no traverse "the wire" in clear text form.
Authorization	» Ensure application has clearly defined the user types and the rights of said users. » Grant only those authorities necessary to perform a given role. » Ensure the Authorization mechanisms work properly, fail securely, and cannot be circumvented. » Do not expose privileged accounts and operations externally.
Cookie Management	» Ensure that unauthorized activities cannot take place via cookie manipulation. » Encrypt the entire cookie if it contains sensitive data. » Ensure secure flag is set to prevent accidental transmission over "the wire" in a non-secure manner. The secure flag dictates that the cookie should only be sent over secure means, such as Secure Sockets Layer (SSL). » Do not store private information on cookies. If required, only store what is necessary.
Data/Input Validation	» All external inputs should be examined and validated.
Error Handling / Information Leakage	» Ensure the application fails in a secure manner. » Ensure resources are released if an error occurs. » Do not expose system errors to the user.

Table 3 – OWASP's Code Review Guide Recommendations

General Principle	Recommendations
Logging/Auditing	» Ensure the payload being logged is of a defined maximum length and the logging mechanism enforces that length. » Log both successful and unsuccessful authentication attempts. » Log access to sensitive data files. » Log privilege escalations made in the application. » Do not log sensitive information.
Cryptography	» Ensure the application is implementing known good cryptographic methods. » Do not transmit sensitive data in the clear, internally or externally. » Do not develop custom cryptography.

Proper Server Configurations

Securing the web application is important, but without properly securing the server configurations, the system is still vulnerable. OWASP's Secure Coding Practices Quick Reference Guide defines the following recommendations for properly securing a server:

- Ensure that the organization has the latest approved versions for the servers, frameworks, and system components.
- Ensure the system provide mechanisms for patching.
- Disable unused HTTP methods, such as the WebDAV extensions. If an extended HTTP method is required for supporting file handling, utilize a well-vetted authentication mechanism.
- 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.
- A software change control system should be employed to manage and record changes to the code both in the development and production phase.
- If the web server handles both HTTP 1.0 and 1.1, ensure that both are configured in a similar manor or insure that any difference that may exist (e.g. handling of extended HTTP methods) are understood.
- Remove unnecessary information from HTTP response headers related to the operating system (OS), web-server version and application frameworks

Secure Session Management

In web applications, sessions are what allow users to use the application while only authenticating themselves once at the beginning of the session. Once a user is authenticated, the application issues a Session ID to ensure that all actions during the session are being performed by the user who originally supplied their authentication information. Attackers often manipulate the Session ID to steal a valid session from an authenticated user. Defense against such attacks include changing a user's Session ID by asking the user to re-authenticate when the session has timed out or when the user attempts to use a functionality that is designated as sensitive. Examples of these attacks and the design principles, from David Rook, are listed in Table 4 below.

Table 4– Session Management Solutions (source: Security Ninja: Security Research, News & Guidance)

Session Management Issue	**How to Avoid It**
Attacker guessing the user's Session ID.	Session IDs should be created with the same standards as passwords. This means that the Session ID should be of considerable length and complexity. There should not be any noticeable pattern in the Session IDs that could be used to predict the next ID to be issued.
Attacker stealing the user's Session ID.	Session IDs, like all sensitive data, should be transmitted by secure means (such as HTTPS) and stored in a secure location (not publically readable).
Attacker setting a user's Session ID (session fixation).	The application should check that all Session IDs being used were originally distributed by the application server.

OWASP provides a Session Management Cheat Sheet for designers to reference while trying to mitigate session management issues. The Cheat Sheet provides different techniques from managing the session ID life cycle to providing a Session Attack Detection. Examples from the Cheat Sheet are provided below:

- » Developers should use the "secure" cookie attribute, which is provided by the cookie, to force the web browser to send the cookie through an encrypted HTTPS (SSL/TLS) connection.
- » Web applications should validate and filter out invalid Session ID values via the client-side before processing them on the server-side.
- » Web applications should renew or regenerated Session IDs after any changes to the privilege level within the associated user session.
- » All sessions should have an expiration time. Expiration times will prevent attackers from reusing valid session ID and hijacking the associated session.
- » Web applications must provide a visible and easily accessible logout (logoff, exit, or close session) button, so the user can manually close the session at any time.
- » For additional client-side defense, use JavaScript to force the user to re-authenticate if a new web browser tab or window is opened against the same web application.
- » Web applications should be able to detect brute force Session ID attacks. This can be accomplished by looking for multiple attempts to gather or use different session IDs from the same IP address(es).
- » To prevent Session hijacking, it is highly recommended that the Session ID binds to the user or client properties, such as the client's IP address, User-Agent, or client-based digital certification.

Transport Layer Protection

The primary benefit of using Transport Layer Security (TLS) and its predecessor, Secure Socket Layer (SSL), is that it protects the web application's data from unauthorized disclosures and modifications when transmitted between the client (web browser) and the web application server. TLS also provides four additional benefits that are often overlooked; it guarantees integrity, confidentiality, replay protection, and end-point authentication.

Providing proper transport layer protection can affect the site's whole design. It is easiest to require TLS for the entire site, but for performance reasons, many websites apply TLS to only private pages or 'critical' pages. The problem with limited TLS protection is that if it is implemented improperly, it can expose session IDs, cookies, and other sensitive data. In order to prevent sensitive data leakage, OWASP provides the recommended minimum settings when applying TLS:

- » Require TLS for all sensitive pages. Non-TLS requests to these pages should be redirected to the TLS page.
- » Set the 'secure' flag on all sensitive cookies. This will prevent the browser from sending any cookie with the 'secure' flag enabled to any 'HTTP' connections.

» Configure your TLS provider to only support strong (e.g. FIPS 140-2 compliant) algorithms.
» Ensure your certificate is valid, not expired, not revoked, and matches all domain used by the site.
» Backend and other connections should also use TLS or other encryption technologies.

Securing the Password Process

The username and password are important forms of authenticating the user, but with weak passwords and poorly developed "forgot-password" features, a user's account can be easily infiltrated. To protect the users from creating weak passwords, passwords should have strongly enforced rules and expectations; OWASP's Secure Coding Practices Quick Reference Guide suggests the following:

» Enforce password complexity requirements established by policy or regulations. Authentication credentials should be sufficient to withstand attacks that are typical of the threats in the deployed environment. An example of password complexity is requiring alphabetic as well as numeric and/or special characters in the password.
» Enforce a minimum length requirement for the password, as established by policy or regulations. OWASP recommends at least eight characters, but sixteen characters or the use of multi-word pass phrases will provide a better solution.
» Enforce password changes based on requirements established in policy or regulations. Critical systems may require more frequent password changes.
» Prevent password re-use; passwords should be at least one day old before they can be changed.
» Disable the account after a certain number of failed login attempts.
» Display generic error messages when a user types in an incorrect username or password.
» Store passwords in the database as salted hash values.

Carefully thought-out security measures for passwords are often broken when a user forgets his or her password. When developing the "forgot-password" feature, Dave Ferguson recommends that the application performs the following procedure before allowing the user to change their password:

1. Ask for the username and hard data. Examples of hard data are the user's last four digits of their social security number, telephone number, birth date, zip code, and/or customer number. The application should display a generic error message, such as "Sorry, invalid data", if the data provided is incorrect.
2. Ask the user to answer two or more of the user's pre-established personal security questions; when developing these security questions, do not allow the user to type in their own questions.
3. Finally, allow the user to reset their password but with password complexity requirements that exist throughout the application, avoid sending the username as a parameter when the form is submitted.

To add additional security to this process, Ferguson recommends that after Step 2, the application sends a randomly-generated verification code (8 or more characters) to the user's phone or email, and then require the user to type the verification code into the application as they perform Step 3 (resetting their password). The downside to this approach is the phone or email stored within the system might be obsolete or invalid.

Preventing Content Injection

Content injection attacks occur when attackers insert malicious code or content into a legitimate site. During a content injection attack, attackers inject malicious content using Cross-Site Scripting (XSS), Cross Site Request Forgery (CSRF), and/or SQL injection. Although many sites are aware of these threats and have programs in place to find and remediate the vulnerabilities, the sheer size and complexity of the websites can make complete remediate of the security weakness implausible.

Content Security Policy (CSP) is one of the many mechanisms used to mitigate broad classes of content injection vulnerabilities. CSP is a declarative policy framework that enables web authors and server administrators the ability to specify the permitted content in their web applications and the ability to restrict the capabilities of that content. CSP provides a method for the server to communicate with the browser, by providing a policy on how content on their website is expected to behave. With proper implementations of CSP, it can mitigate and detect content injection attacks such as XSS and CSRF.

In order to use CSP, the browser must first opting-in by sending a custom HTTPS header, then the server sends the browser a whitelist of content the server allows, and finally the browser disables all JavaScript and allows only content that are on the whitelist. To protect other users, the browser reports back to the server any content in violation of the CSP. The server than can block this content or remove it from the site. There is another option for developers if they do not wish to block any content. The site developer can allow all content on the user's browsers but require the browsers to report back with a list of content received which were not on the whitelist.

Resources

» "Details of a Real Data Breach." Oliver Wai and Bob Matlow, Barracuda Networks, 9 April 2011. <http://www.barracudanetworks.com/ns/downloads/White_Papers/Barracuda_Web_Application_Firewall_WP_Real_Data_Breach.pdf>.

» "OWASP Code Review Guide: 2008 V1.1." OWASP, 2008. <https://www.owasp.org/images/2/2e/OWASP_Code_Review_Guide-V1_1.pdf>.

» "OWASP Secure Coding Practices: Quick Reference Guide." OWASP, November 2010. <https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf>.

» "Session Management." Rook, David, Security Ninja: Security Research, News & Guidance. Realex Payments, 1 December 2010. <http://www.securityninja.co.uk/secure-development/session-management>.

» "Session Management Cheat Sheet." Siles, Raul, OWASP, October 2011. <https://www.owasp.org/index.php/Session_Management_Cheat_Sheet>

» "Transport Layer Protection Cheat Sheet." Michael Coates, Dave Wichers, Michael Boberski, and Tyler Reguly, OWASP ,<https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet>.

» "Best Practices for a Secure "Forgot Password" Feature." Ferguson, Dave, Fishnet Security, 2010. <http://www.fishnetsecurity.com/Resource_/PageResource/White_Papers/FishNetSecurity_SecureForgotPassword.pdf>.

» "Content Security Policy." Brandon Sterne, 2011. <http://www.w3.org/TR/CSP/>.

Embedded Systems Security

According to Bessin, an embedded system is “a software system that must be designed on a platform different from the platform on which the system is intended to be deployed.” The first “platform” stated, refers to an operating system capable of hosting software development, such as Windows, Solaris, HP, etc. This implies that an embedded system is any computer system other than a desktop, laptop, or a mainframe computer. According to Barr of netrino.com, “[o]f the nine billion processors manufactured in 2005, less than 2% became the brains of new PCs, Macs, and UNIX workstations.” Many electronic devices are or have embedded systems. Embedded systems are typically designed to have optimized performance on a dedicated task, involve hard or soft real-time requirements, and are limited by system resources. These requirements/restrictions can cause complications when it comes to implementing security in the architecture and design.

Secure Embedded Design

Designers quickly encounter hardware limitations when developing the software of an embedded system. Many embedded systems are severely constrained by the environment it runs on. With all the limitations and requirements, the security features in an embedded system should be application dependent. When considering security in the architecture and design of an embedded system, there are additional constraints designers must incorporate:

- **Limited Battery Capacity** – many embedded systems are designed to maintain a long battery life, so with heavy computation of cryptographic algorithms, the battery’s life expectancy will be lowered.
- **Processing Capability** – some embedded system architectures are not capable of keeping up with computational demands, such as network routers and web servers, where increasing data rates and complexity of security protocols are required. Tailoring the application to work with security in the architecture and design stage can help mitigate this barrier.
- **Cost** – the cost of an embedded system is one of the most fundamental factors that influence the security architecture. Additional single-chip cryptographic modules can add a higher standard of security on the embedded system, but design and incorporation of the module can be costly.
- **Availability** – the embedded system can be a critical part of a system, such as a nuclear reactor’s control system, which cannot fail or shut down for a long period of time.

Availability can be a vital aspect of the embedded system; the following are examples that designers should incorporate in the system to ensure the system’s availability:

- Use virtual machine’s (VM) “ sandboxing” of unsafe/non-secure components during system operation to isolate less trustworthy components and prevent their flaws from impacting the safe and secure operation of the system overall.
- Develop Multiple Independent Levels of Security and Safety (MILS), which will be discussed later in this pocket guide, to provide separation and virtualization techniques.
- Implement watchdog timers to check for the “I’m alive” signals from processes and use exception handling logic to correct actions before a failure can occur.
- Remove unused functions/code. Unused functions/code can be triggered by unanticipated events, such as unexpected inputs that are misinterpreted by the system.

Most embedded systems are coded in C and C++ due to of the efficiency of the language and the hardware requirements of the system. However, C and C++ do not provide inherent safety measures for vulnerabilities such as buffer overflow and array bound errors. Secure coding guidelines should be followed when developing the code for an embedded system. The Motor Industry Software Reliability Association (MISRA) C/C++ guidelines and the Secure Coding pocket guide are a couple of resources that provide proper coding guidelines for the C and C++ language.

Multiple Independent Levels of Security/Safety (MILS)

Multiple Independent Levels of Security/Safety (MILS) is a high-assurance security architecture for executing different security-level processes on the same high-assurance system. The goal of MILS is to simplify the specification, design, and analysis of security mechanisms. The MILS approach is based on the concept of separation, which tries to provide an environment which is indistinguishable from a physically isolated system. A hierarchy of security services can be obtained through separation. In this hierarchy, each level uses the security services of a lower level to provide a new security functionality on the higher levels. Each level is responsible for its own security domain and nothing else.

The MILS architecture was created to simplify the process of the specification, design, and analysis of high-assurance computing systems. The MILS architecture breaks systems function into three levels: the application layer, the middleware service layer, and the separation kernel.

The MILS separation kernel (SK), sometimes called the partitioner layer, is the base layer of the system. The SK is responsible for enforcing data separation and information flow control; providing both time and space partitioning. The SK should provide the following:

» **Data Separation** – the memory address space, or objects, of a partition should be completely independent of other partitions. This separation helps isolate security critical code from non-security critical code, which in turn allows developers to focus more on verifying the security critical partitions.

» **Information Flow** – pure data separation is not practical so there is a need for the partitions to communicate with each other. The SK will define precise moderated mechanisms for inter-partition communication.

» **Sanitization** – the SK is responsible for cleaning any shared resources (registers, system buffers, etc.) before allowing a process in a new partition to use them.

» **Damage Limitation** – address spaces of partitions are separated, so faults or security breaches in one partition are limited by the data separation mechanism.

Resources

» "Embedded Systems: A Primer." Bessin, Geoffrey, IBM, 24 November 2003. <http://www.ibm.com/developerworks/rational/library/806.html>.

» "Embedded Systems Glossary." Barr, Michael, Netrino, <http://www.netrino.com/Embedded-Systems/Glossary>.

» "Execution Partitioning for Embedded Systems Increase Security, Reliability." Schwaderer, Curt, Software Corner, June 2006. <http://pdf.cloud.opensystemsmedia.com/advancedtca-systems.com/software-corner-2006,06.pdf>.

» "Safety and Security Considerations for Component-Based Engineering of Software-Intensive Systems." Goertzel, Karen Mercedes, *et al.,* Booz Allen Hamilton, 7 February 2011. <https://buildsecurityin.us-cert.gov/swa/downloads/NAVSEA-Composition-DRAFT-061110.pdf>.

» "Practical Embedded Security: Building Secure Resource-Constrained Systems." Stapko, Timothy,

» "Security in Embedded Systems: Design Challenges." Ravi, Srivaths, et al., 3 August 2004. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.83.586&rep=rep1&type=pdf>.

Database Security

Data can be a company's greatest asset, so securing it should be the company's top priority. In order to secure the data, it is necessary to properly architect and design the application, the database, and the communication between the application and the database. Many attackers tend to aim for the data in a software system; this is why SQL injections (CWE-89) have become a common issue with database-driven websites. Input validation techniques can help mitigate SQL injections and data theft, but additional constraints should be imposed on the application's interactions with the database. OWASP recommends that the application should:

» Use the lowest level of privileges when accessing the database.

» Connect to the database with different credentials for every trust distinction.

» Store connection strings (e.g. username, password, and the name of the database) in a separate encrypted configuration file and on a trusted system; connection strings should not be hard-coded within the application. Protocols, such as OAuth, can then be used to properly authenticate between the trusted system and the database.

» Close the connection as soon as possible.

» Use strongly typed parameterized queries.

» Prevent the execution of the database command if either input validation or output encoding fails. Refer to the Secure Coding pocket guide for additional information on input validation and output encoding.

OWASP also recommends the use of secure credentials when accessing the database. The architecture and design of the credentials will often affect the whole application, due to its relationship with the privilege of the user. An example of secure credentials is the United States Department of Agriculture (USDA) eAuthentication Service. This authentication service separates the users in four different levels, with each level requiring a higher proof of identification from the user. The higher the user's level, the more privileges they have to access and write to the data. Secure credentials help prevent attackers from automating the creation of accounts and prevent these accounts from having significant access to the database.

Just securing the application is not sufficient; extra measures should also be taken in the design of the database.   When designing the database, be sure to:

» Keep the database server separate from the application server.   For a higher level of security, use different communication protocols than TCP/IP between application and database servers..

» 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 to reduce the attack surface) and remove unnecessary vendor contents (e.g., sample schemas).

» Remove all default database administrative passwords.

» Utilize strong passwords/phrases or implement multi-factor authentication.

» Disable default accounts that are not required to support business requirements.

» Enforce access control effectively and authenticate clients stringently.

» Audit sensitive information and keep audited information manageable.

» Ensure that variables are strongly typed.

» Use stored procedures to abstract data access and allow for the removal of permissions to the base tables in the database.

Resources

- » "OWASP Secure Coding Practices: Quick Reference Guide." OWASP, November 2010. <https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf>.
- » "USDA eAuthentication Service." United States Department of Agriculture. 4 November 2004. <http://www.ocio.usda.gov/directives/doc/DR3610-001.pdf>.

Mobile Applications

Mobile Applications, like any other application, need a secure design and architecture in order to mitigate security risks. Creating a secure architecture and design can be challenging for mobile applications because of their complicated threat model. Because an attacker can access a stolen device anytime, the exposure of a mobile device is much higher than a desktop. Also shifting between the calls and the wireless networks, the mobile device often requires more complex connectivity, thus complicating their threat models.

All conventional secure design and coding principles are relevant to mobile applications. For example:

- » Use secure coding guidelines to protect against the security risks in the native code, such as buffer overflows and format string vulnerabilities.
- » Positively validate data coming from third-party web services and other untrusted sources before integrating into the application.
- » Use platform-specific permission controls to provide only the minimum sets of required capabilities for an application to accomplish its intended use. This applies to access of capabilities such as reading the GPS location of the device, taking photos, or recording audio.

Mobile devices are particularly vulnerable to attacks if lost, left unattended, or stolen. A skilled attacker with physical access to the mobile device can access raw data and control the execution of software on the device. This gives the attackers the ability to steal sensitive data (such as passwords), recover source code, steal intellectual property, and identify network services that are used to support the mobile application. While devices can often be disabled remotely, there is still a risk between the time the device is compromised and the time it is disabled. When designing mobile applications, developers can do the following to mitigate these risks:

- » Avoid writing personally identifiable information (PII) and other sensitive data to the device when possible. If this cannot be avoided:
    - o Ensure that all data written to the device is encrypted. Like encrypted hard drives, all the data should be encrypted for maintainability.
    - o Pay particular attention to the management of encryption keys. Applications storing encryption keys on the device alongside the encrypted data do not provide proper security; attackers will likely be able to recover the encrypted data, the keys used to encrypt the data, and the application routines explaining how the data was encrypted.
- » Never store extremely sensitive information such as passwords and credit card information on the device.
- » Store all data on fixed platforms that are under your administrative control.
- » Do not allow sensitive files to be world readable and writable. Platform-provided file protections will not stop attackers with physical access to the device, but it may be effective in guarding against attacks from other malicious applications installed on the device.
- » Do not install applications (OAuth for example) that assume the mobile platform is a trusted environment.

The physical security of the mobile device must be considered when sending data to web applications. Some application programming interfaces (APIs), such as the Geolocation API, give the web application the ability to read the mobile device's GPS location, with the user's permission. This capability opens sensitive data on the mobile device to the web, allowing attackers to maliciously use this data to track certain users and steal sensitive information. With growing capabilities of HTML/JavaScript and mobile device interaction, web applications will eventually have full control of the mobile device. If the mobile device must provide information to the web service, take appropriate security measures such as alerting the user, deleting the information after usage, and providing only non-sensitive information.

Conversely, mobile applications can also introduce a new attack vector to web servers. Attackers with access to a device will likely be able to extract information about the web services supporting the application as well as the details of the communication protocols used between the device and the web services. If a mobile application communicates with a web server or any other network resources, both the server and the application should be properly authenticated before accessing sensitive services.

In addition to using authentication, the authentication process should be properly architected, designed, and maintained. An example of an architectural flaw would be using the OAuth Core 1.0 protocol on the mobile platform to authenticate across services. When authenticating the user to a remote trusted server, OAuth assumes the host is a trusted platform. Because an attacker can readily obtain control of the device, the assumption of trust is inappropriate. With OAuth running on a mobile device, an attacker can use the OAuth authentication process to gain access to the Protected Resources not only on the mobile device but also on the host servers. This is because OAuth has the token and the shared secret key in the application, on the mobile device. Any Cross-Site Request Forgery (CSRF) protection does not mitigate this attack. The best mitigation practice is to assume the mobile device is an untrusted source; do not use protocols or develop applications that assume the device is trusted.

Sandboxing is available on many mobile operating systems (OS), such as Android and iOS. Sandboxing can prevent malicious applications from accessing and modifying the mobile system or the data of another application. While sandboxing is available to prevent malicious application attacks, there are ways around this, such as the Intent class provided by the Android OS. The Intent class allows the current application the ability to launch another application. This can be used harmfully by an attacker to launch malicious websites. The best mitigation techniques are to sandbox the application and validate the data retrieved from other applications.

Mobile devices are often connected to a variety of both trusted and untrusted communication networks. Securing sensitive information as it is communicated wirelessly to and from the mobile devices is vital. There are many ways to do this such as enforcing encryption, digital signatures, or using a Virtual Private Network (VPN). Applications relying on HTTPS to secure network communications should be sure to verify the identity of servers through the use of certificates to prevent man-in-the-middle attacks. Unlike other kinds of applications, mobile applications are usually on devices with battery life constraints and are more likely to be used in areas with low bandwidth network access. Because of this, care must be taken to ensure that the process of securing the application's transmission is done as efficiently as possible.

Mobile Risks

OWASP has developed a list of Top 10 Mobile Risks to help designers develop a better threat model for mobile applications. These risks are intended to be platform-agnostic and focus on areas of risk rather than individual vulnerabilities. The following are the risks and their brief descriptions:

1. Insecure Data Storage – Includes: storing data on the device unencrypted, caching data not intended for long-term storage, weak or global permissions, and not leveraging platform best-practices.
    - Prevention Tips: Store only what is absolutely required; never use public storage areas (e.g. SD cards); leverage secure containers and platform provided file encryption APIs; and do not grant files world readable or world writeable permissions.
2. Weak Server Side Controls – This applies to the backend services, which cannot trust the client. Existing controls may need to be re-evaluated (e.g. out of band communications).
    - Prevention Tips: Understand the additional risks mobile applications can introduce into existing architectures and use the wealth of knowledge already out there (e.g. OWASP Web Top 10, Cloud Top 10, Cheat Sheets, Development Guides).

3. Insufficient Transport Layer Protection – Includes: complete lack of encryption for transmitted data; weakly encrypted data in transit; and/or strong encryption but security warnings were ignored.
    - Prevention Tips: ensure all sensitive data leaving the device is encrypted, this includes data over carrier networks, WiFi, etc.
4. Client Side Injection – With the mobile phone on the client side of a web application, there are still some familiar faces/attacks such as XSS and SQL injection, but in addition there are new twists, such as abusing the phone dialer or SMS and abusing the in-app payments.
    - Prevention Tips: Sanitize or escape untrusted data before rendering or executing it; used prepared statements for database calls (concatenation is a bad practice); and minimize the sensitive native capabilities tied to hybrid web functionality.
5. Poor Authorization and Authentication – Poor implementation of the authorization and authentication on a mobile device can cause unauthorized access and privilege escalation by the users.
    - Prevention Tips: Contextual information can enhance the authentication process but only as a part of a multi-factor authentication; never use device ID or subscriber ID as a sole authenticator; and authenticate all API calls to paid resources.
6. Improper Session Handling – Mobile sessions are generally longer since it is convenient and provides better usability.
    - Preventions Tips: Do not use device identifier as a session token. Make users re-authenticate every so often and ensure that tokens can be revoked quickly in the event of a stolen/lost device.
7. Security Decisions via Untrusted Inputs – Improper security decisions via untrusted inputs can result in attackers bypassing permissions and security models. This is similar but different depending on the platform, iOS can have their URL Schemes abused and Android can have their Intents abused.
    - Prevention Tips: Check caller's permissions at input boundaries; prompt the user for additional authorization before allowing consummation of paid resources; when permission checks cannot be performed, ensure additional steps are required to launch sensitive actions.
8. Side Channel Data Leakage – Mixing of not disabling platform features and programmatic flaws can leave sensitive data in unintended places such as web caches and screenshots.
    - Prevention Tips: understand what 3rd party libraries in your application is doing with the user data; never log credentials, PII, or other sensitive data to system logs; remove sensitive data before screenshots are taken; before releasing apps, debug them to observe files created, written to, or modified in any way; and test you application across as many platform versions as possible.
9. Broken Cryptography – Encoding, obfuscation, and serialization is not considered encryption.
    - Preventions Tips: do not store the key with the encrypted data; use what your platform already provides; and do not develop in-house cryptography.
10. Sensitive Information Disclosure – Applications can be reversed engineered with relative ease, obfuscation raises the bar, but does not eliminate the risk. Reverse engineering of the application can open up API keys, passwords, and sensitive business logic.
    - Preventions Tips: Do not store the private API keys in the client; keep proprietary and sensitive business logic on the server; and never hardcode the password.

Resources

» "Mobile Application Security: Promises and Pitfalls in the New Computing Model." Stamos, Alex, iSECPartners.com. iSEC Partners. 1 December 2010. <http://www.cio.ca.gov/OIS/Government/events/documents/Mobile_Application_Security.pdf>.

» "Smart Mobile: Five Things for Smart Mobile Application Developers to Know About Security." Cornell, Dan, Machine to Machine, 16 August 2010. <http://m2m.tmcnet.com/topics/smart-mobile/articles/95242-five-things-smart-mobile-application-developers-know-security.htm>.

» "OAuth Security Advisory: 2009.1." OAuth, 23 April 2009. <http://oauth.net/advisories/2009-1/>.

» "Inside OAuth." Steve Gibson and Leo Laporte, Security Now!, 15 September 2010. <http://www.grc.com/sn/sn-266.pdf>.

» "Security and Permissions." Android Developers. 20 May 2011. <http://developer.android.com/guide/topics/security/security.html>.

» "OWASP Top 10 Mobile Risks." OWASP. <http://www.slideshare.net/JackMannino/owasp-top-10-mobile-risks>

Formal Methods and Architectural Design

Formal methods are the incorporation of mathematically based techniques for the specification, development, and verification of software. Formal methods can be used to improve software security but can be costly and also have limitations of scale, training, and applicability. To compensate for the limitations of scale, formal methods can be applied to selected parts or properties of a software project, in contrast to applying them to the entire system. As for training limitations, it may be difficult to find developers with the needed expertise in formal logic, the range of appropriate formal methods for an application, or appropriate automated software development tools for implementing formal methods, therefore formal methods are best applied on reusable software. Formal methods does not apply to just software, but can be useful for verifying a system. Verification shows that each step in the development satisfies the requirements imposed by previous steps.

Formal methods can be used in the design phase to build and refine the software's formal design specification. Since the specification is expressed in mathematical syntax and semantics, it is precise in contrast to non-formal and even semi-formal specifications that are open to reinterpretation.

There are many specification languages available, but each language is limited in scope to the specification of a particular type of software or system, *e.g.,* security protocols, communications protocols, and encryption algorithms. Examples of specification languages and type include:

» **Model-Oriented**: Z, Vienna Development Method (VDM);
» **Constructive**: Program/Proof Refinement Logic (PRL);
» **Algebraic/Property-Oriented**: Larch, Common Algebraic Specification Language (CASL), OBJ;
» **Process Model**: Calculus of Communicating Systems (CCS), Communicating Sequential Processes (CSP); and
» **Broad Spectrum**: Rigorous Approach to Industrial Software Engineering (RAISE) Specification Language (RSL), LOTOS (language for specifying communications protocols).

Formal Methods in Architectural Design – Formal methods can be used in the architecture phase to:

» Specify architectures, including security aspects of an architectural design;

- Verify that an architecture satisfies the specification produced during the previous phase, if that specification itself is in a formal language;
- Establish that an architectural design is internally consistent;
- Automatically generate prototypes; and
- Automatically generate a platform-dependent architecture.

Information Assurance (IA) applications frequently must meet mandatory assurance requirements, and the use of formal methods for IA applications are more prevalent than for many other types of applications. Formal methods are used in assuring IA applications can be used to assure correctness for those willing to incur the costs. In IA applications, formal methods have been used to prove correctness of security functionalities for authentication, secure input/output, mandatory access control and security-related trace properties such as secrecy.

A variety of automated tools are available to assist developers in adopting formal methods. Theorem provers are used to construct or check proofs. Theorem provers differ in how much the user can direct them in constructing proofs. Model checkers are a recent class of theorem provers that has extended the practicality of formal methods. Another range of automated tools are associated with model-driven architecture (MDA) and model-driven development (MDD). These are considered semiformal rather than formal methods.

The State-of-the-Art Report describes how, in *Correctness by Construction,* Anthony Hall and Roderick Chapman depict the development of a secure Certificate Authority, an IA application. The formal top-level specification (architecture design) was derived from the functionality defined in the user requirements, constraints identified in the formal security policy model, and results from the prototype user interface. Praxis used a type checker to automatically verify the syntax in the formal top-level specification and reviews to check the top-level specification against the requirements. The formal security policy model and the formal top-level specification are written in Z, a formal specification language, while the detailed design derived from the top-level specification is written in CSP.

In *Modeling and Analysis of Security Protocols,* Peter Ryan *et al*, describe their use of Failure Divergence Refinement (FDR), a model-checking tool available from Formal Systems Ltd., the Caspar compiler, and CSP. They use these tools to model and analyze a protocol for distributing the symmetric shared keys used by trusted servers and for mutual entity authentication.

Other applications of formal methods are mentioned in *Security in the Software Life Cycle*. These include applications by Kestrel and Praxis where they describe technology that includes Software specification, Language, Analysis, and Model-checking (SLAM), which is Microsoft's model checking tool, the Standard Annotation Language (SAL), and Fugue.

Formal Methods and Detailed Design – The formal methods used in detailed design and implementation usually are different from those used in system engineering, software requirements, and software architecture. Formal methods adopted during earlier phases of the SDLC support the specification of systems and system components and the verification of high-level designs. For architecture design, organizations use model checkers such as VDM, and formal specification languages such as Z. Formal methods commonly used in detailed design and implementation are typically older methods, such as Edsger Dijkstra's predicate transformers and Harlan Mill's functional specification approach.

Formal methods for detailed design are most useful for:

- Verifying the functionality specified formally in the architecture design phase is correctly implemented in the detailed design or implementation phases; and
- Documenting detailed designs and source code.

For example, under Dijkstra's approach, the project team would document a function by specifying pre- and post-conditions. Preconditions and post-conditions are predicates such that if the precondition correctly characterizes a program's state on entry to the function, the post-condition is established upon exiting. An invariant is another important concept from this early work on formal methods. An invariant is a predicate whose truth is maintained by each execution of a loop or for all uses of a data structure. A possible approach to documentation includes stating invariants for loops and abstract data types. Without explicit and executable identification of preconditions, post-conditions, and invariants for modules, formal methods in detailed design are most appropriate for verifying correctness when the interaction between system components is predefined and well-understood.

Resources

- “Software Security Assurance: A State-of-the-Art Report”(SOAR). Goertzel, Karen Mercedes, *et al*., Information Assurance Technology Analysis Center (IATAC) of the DTIC. 31 July 2007. <http://iac.dtic.mil/iatac/download/security.pdf>.

Design Review and Verification

What is an assurance case? Assurance cases use a structured set of arguments and a corresponding body of evidence to demonstrate that a system satisfies specific claims with respect to its security properties.

Design reviews should be performed by multiple persons with relevant software security expertise and should represent a broad cross-section of stakeholder interests. Formal techniques available include scenario-based reviews that were created for architecture reviews. Reviews including security issues are essential at all levels of design. An independent, third-party review is recommended whenever possible. Design-related portions of the assurance case should be reviewed as well, since the best results occur when one develops much of the design assurance case along with the design. Checklists on verification requirements are also useful, such as the one OWASP provides from their Application Security Verification Standard 2009 – Web Application Standard document.

Conducting verification activities during the design phase includes:

- Structured inspections, conducted on parts or views of the high-level design throughout the design phase;
- Independent verification and validation (IV&V) reviews;
- A preliminary design review conducted at the end of the architecture design phase and before entry into the detailed design phase; and
- A critical design review added at designated points in the software development life cycle.

Design Verification–The design should be verified considering the following criteria:

- The design is correct and consistent with and traceable to requirements;
- The design implements the proper sequence of events, inputs, outputs, interfaces, logic flow, allocation of timing and sizing budgets, error definition, isolation, and recovery;
- The selected design can be derived from requirements; and
- The design implements safety, security, and other critical requirements correctly as shown by suitably rigorous methods.

The decision on which reviews to conduct and their definitions usually takes place during the evolution of the development schedule. Such definitions typically include entry criteria, exit criteria, the roles of participants, the process to be followed, and data to be collected during each review. The choice of reviews, particularly those performed as part of IV&V, is partly guided by the evaluation requirements at the Common Criteria (CC) Evaluation Assurance Level (EAL). The CC EALs provide an increasing scale that balances the level of assurance obtained with the cost and feasibility of acquiring that degree of assurance. The processes for reviewing the architecture and detailed design should also accommodate later reviews of architecture and design modifications. Penetration testing is performed to determine the security sufficiency of the software architecture and design.

Resources

- "Software Security Assurance: A State-of-the-Art Report"(SOAR). Goertzel, Karen Mercedes, *et al.*, Information Assurance Technology Analysis Center (IATAC) of the DTIC. 31 July 2007. <http://iac.dtic.mil/iatac/download/security.pdf>.
- "BuildSecurityIn: Assurance Cases." United States Government. Department of Homeland Security, <https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/assurance.html>.
- "OWASP Application Security Verification Standard 2009 – Web Application Standard." OWASP, June 2009. <http://www.owasp.org/images/4/4e/OWASP_ASVS_2009_Web_App_Std_Release.pdf>.
- "Common Criteria for Information Technology, Part3: Security assurance components." September 2007. <http://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R2.pdf>

Key Architecture and Design Practices for Mitigating Exploitable Software Weaknesses

The Common Weakness Enumeration (CWE) [see http://cwe.mitre.org] defines a unified, measurable set of software weaknesses. CWE enables effective discussion, description, selection, and use of software security tools and services to identify weaknesses in source code and operational systems. CWE also enhances understanding and management of software weaknesses related to architecture and design. Mitigation of the most egregious exploitable weaknesses can be best addressed early in the software development life cycle, such as the architecture and design phase. Some architecture and design security issues are directly related to secure coding, please refer to the Secure Coding pocket guide for additional information.

What evidence substantiates that the architecture and design of the software will be secure? Development teams should provide that assurance at appropriate phases as set forth below:

Architecture Phase:

- **Control filenames**: When the set of filenames is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames, and reject all other inputs. For example, ID 1 could map to "inbox.txt" and ID 2 could map to "profile.txt". Features such as the ESAPI AccessReferenceMap provide this capability. CWE-73
- **Enforce boundaries**: Run the code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict all access to files within a particular directory. Examples include the Unix chroot jail and AppArmor. CWE-73
- **Place server-side checks**: Ensure any security checks performed on the client-side are duplicated on the server-side. Attackers can bypass the client-side checks by modifying values after the checks have been performed or by changing the client to remove the client-side checks entirely. If the server receives input that should have been rejected by the client, then it may be an indication of an attack. CWE-602
- **Assign permissions carefully**: When using a critical resource such as a configuration file, check to see if the resource has insecure permissions (such as being modifiable by any regular user), and generate an error or exit the software if there is a possibility that the resource has been modified by an unauthorized party. Reduce the attack surface by carefully defining distinct user groups, privileges, and/or roles. CWE-732
- **Provide orderly resource-shutdown**: Use a language with features that can automatically mitigate or eliminate resource-shutdown weaknesses. For example, languages such as Java, Ruby, and Lisp perform automatic garbage collection that releases memory for objects that have been de-allocated. CWE-404
- **Protect against inappropriate initialization**: Use a language with features that can automatically mitigate or eliminate weaknesses related to initialization. For example, in Java, if the programmer does not explicitly initialize a variable, the code could produce a compile-time error (if the variable is local) or automatically initialize the variable to the default

value for the variable's type. Identify all variables and data stores that receive information from external sources, and apply input validation to make sure that they are only initialized to expected values. CWE-665

» **Restrict executables**: Run the code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system, which may effectively restrict which code can be executed. Examples include the Unix chroot jail and AppArmor. CWE-94

Design Phase:

» Preserve OS command structure to mitigate risk OS command injection. CWE-78
» **Use modular cryptography**: Design the software so that one cryptographic algorithm can be replaced with another, and do not develop custom cryptographic algorithms. Specify the use of languages, libraries, or frameworks that make it easier to use strong cryptography. (Consider the ESAPI Encryption feature.) CWE-327
» Encrypt data, with a reliable encryption scheme, before transmitting sensitive information. CWE-319
» Specify the download of code only after integrity check. CWE-494
» **Protect against Denial of Service attacks**: Mitigate race conditions and minimize the amount of synchronization necessary to help reduce the likelihood of a denial of service where an attacker may be able to repeatedly trigger a critical section by using synchronization primitives (in languages that support it). Only wrap these around critical code to minimize the impact on performance. Use thread-safe capabilities such as the data access abstraction in Spring. CWE-362
» **Protect against CSRF**: Mitigate risk from Cross-Site Request Forgery (CSRF) by using the ESAPI Session Management control and anti-CSRF packages such as the OWASP CSRFGuard. Identify especially dangerous operations. When the user performs a dangerous operation, send a separate confirmation request to ensure that the user intended to perform that operation. CWE-352
» **Authenticate**: If some degree of trust is required between the two entities, then use integrity checking and strong authentication to ensure that the inputs are coming from a trusted source. Design the product so that this trust is managed in a centralized fashion, especially if there are complex or numerous communication channels. CWE-602
» Preserve web page structure to mitigate risk from cross-site scripting. CWE-79

Resource

» "Key Practices for Mitigating the Most Egregious Exploitable Weaknesses." SwA Pocket Guide Series. DHS Build Security In (BSI) portal, 24 May 2009. <https://buildsecurityin.us-cert.gov/swa/pocket_guide_series.html>.

Questions to Ask Developers

Managers in development and procurement organizations should ask questions to determine if the team responsible for delivering the software uses practices that harness secure architecture and design in developing secure software. These questions should highlight the major architecture and design considerations for assuring secure applications. For a more comprehensive set of questions, reference the “Software Assurance in Acquisition and Contract Language” and “Software Supply Chain Risk Management & Due-Diligence” Pocket guides. The following are example of questions relating to each section in this pocket guide:

- How does the architect determine how to protect the software from attackers? Is the architect using attack trees, threat models, and/or misuse/abuse case?
- Are misuse cases utilized during the design process? If so, what modeling techniques are used to define and handle misuse cases?
- Are the architects using secure software design principles to mitigate potential weaknesses? If so, where are they obtaining these principles?
- Are secure design patterns being used?
- Is the software using known, good cryptographic methods/algorithm?
- How does the software perform access control checks?
- How does the application ensure that only authorized personnel are accessing sensitive information?
- How does the software assure that data transmitted over a communications channel cannot be intercepted and/or modified?
- Are formal methods incorporated into the architectural design to improve software security? If so, what methods are being used and why?
- How does the server-side security check for bypasses in the client-side validation? What techniques are being used to prevent this?
- How does the server ensure that the Session ID has not been stolen or manipulated?
- If the application requires a username and password, are the password complexity requirements following company’s policy and/or regulations? If the user forgets their password, is the forgotten-password procedure strict and secure?
- If the software is running on an embedded system, is the system following a high-assurance security architecture, such as MILS?
- Are connection strings, used to connect to the database, stored in a separate, encrypted configuration file? Is the configuration file on a trusted system?
- Is the application using strongly typed, parameterized queries when accessing the database?
- Does the mobile application mitigate the risks listed on OWASP’s Top 10 Mobile Risks list?
- How does the team prevent weaknesses in the initialization of variables and objects?
- How does the software assure that the state information has not been tampered with?
- How does the software prevent attackers from repeatedly triggering a synchronized critical section?

Conclusion

This pocket guide summarizes key architecture and design techniques for software security and offers guidance on when they should be employed during the Software Development Life Cycle (SDLC). It focuses on the practices and knowledge required to establish the architecture and high-level design for secure software during the SDLC. Mitigation of the most egregious exploitable weaknesses can be best addressed early in the SDLC in the architecture and design phase. Some security issues are not apparent in the coding phase but can be identified during the design phase via threat models and misuse cases. When developing web applications, Session ID should be protected with length and complexity. With mobile applications, developers should consider encrypting sensitive information and transmitting data securely. Formal methods can improve software security with mathematically proven techniques. Sometimes a hierarchy of security services can be the best option like Multiple Independent Levels of Security and Safety (MILS). Lastly, managers should ask questions relating to the CWE to assure them that development teams are taking appropriate measures to prevent software weaknesses. In summary, the materials and resources provided by this guide can be used as a starting point for developing secure architecture and design techniques.

The Software Assurance Pocket Guide Series is developed in collaboration with the SwA Forum and Working Groups and provides summary material in a more consumable format. The series provides informative material for SwA initiatives that seek to reduce software vulnerabilities, minimize exploitation, and address ways to improve the routine development, acquisition and deployment of trustworthy software products. Together, these activities will enable more secure and reliable software that supports mission requirements across enterprises and the critical infrastructure.

For additional information or contribution to future material and/or enhancements of this pocket guide, please consider joining any of the SwA Working Groups and/or send comments to Software.Assurance@dhs.gov. SwA Forums and Working Group Sessions are open to all participants and free of charge. Please visit https://buildsecurityin.us-cert.gov for further information.

Resources

» "National Information Assurance (IA) Glossary: CNSS Instruction No. 4009." Committee on National Security Systems, 26 April 2010. <http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf>.
» "Glossary of Key Information Security Terms." National Institute of Standards and Technology (NIST), February 2011. <http://csrc.nist.gov/publications/nistir/ir7298-rev1/nistir-7298-revision1.pdf>.
» "What is CWE?" MITRE, 26 September 2007. <http://cwe.mitre.org/about/index.html>.
» "Terminology." MITRE, 17 July 2007. <http://cve.mitre.org/about/terminology.html>.
» "About Software Assurance." DHS Build Security In (BSI) portal, <https://buildsecurityin.us-cert.gov/swa/about.html>.

No Warranty

This material is furnished on an "as-is" basis for information only. The authors, contributors, and participants of the SwA Forum and Working Groups, their employers, the U.S. Government, other participating organizations, all other entities associated with this information resource, and entities and products mentioned within this pocket guide make no warranties of any kind, either expressed or implied, as to any matter including, but not limited to, warranty of fitness for purpose, completeness or merchantability, exclusivity, or results obtained from use of the material. No warranty of any kind is made with respect to freedom from patent, trademark, or copyright infringement. Reference or use of any trademarks is not intended in any way to infringe on the rights of the trademark holder. No warranty is made that use of the information in this pocket guide will result in software that is secure. Examples are for illustrative purposes and are not intended to be used as is or without undergoing analysis.

Reprints

Any Software Assurance Pocket Guide may be reproduced and/or redistributed in its original configuration, within normal distribution channels (including but not limited to on-demand Internet downloads or in various archived/compressed formats).

Anyone making further distribution of these pocket guides via reprints may indicate on the back cover of the pocket guide that their organization made the reprints of the document, but the pocket guide should not be otherwise altered. These resources have been developed for information purposes and should be available to all with interests in software security.

For more information, including recommendations for modification of SwA pocket guides, please contact Software.Assurance@dhs.gov or visit the Software Assurance Community Resources and Information Clearinghouse: https://buildsecurityin.us-cert.gov/swa to download this document either format (4"x8" or 8.5"x11").

Software Assurance (SwA) Pocket Guide Series

SwA is primarily focused on software security and reliability to mitigate risks attributable to exploitable software; better enabling resilience in operations. SwA Pocket Guides are provided; with some yet to be published. All are offered as informative resources; not comprehensive in coverage. All are intended as resources for 'getting started' with various aspects of software assurance. The planned coverage of topics in the SwA Pocket Guide Series is listed:

SwA in Acquisition & Outsourcing

I. Software Assurance in Acquisition and Contract Language
II. Software Supply Chain Risk Management & Due-Diligence

SwA in Development

I. Integrating Security in the Software Development Life Cycle
II. Key Practices for Mitigating the Most Egregious Exploitable Software Weaknesses
III. Software Security Testing
IV. Requirements Analysis for Secure Software
V. Architecture & Design Considerations for Secure Software
VI. Secure Coding
VII. Security Considerations for Technologies, Methodologies & Languages

SwA Life Cycle Support

I. SwA in Education, Training & Certification
II. Secure Software Distribution, Deployment, & Operations
III. Code Transparency & Software Labels
IV. Assurance Case Management
V. Assurance Process Improvement & Benchmarking
VI. Secure Software Environment & Assurance Ecosystem
VII. Penetration Testing throughout the Life Cycle

SwA Measurement & Information Needs

I. Making Software Security Measurable
II. Practical Measurement Framework for SwA & InfoSec
III. SwA Business Case

SwA Pocket Guides and related documents are freely available for download via the DHS NCSD Software Assurance Community Resources and Information Clearinghouse at https://buildsecurityin.us-cert.gov/swa.