

## Software Architecture/Security

### Planning for an Assurable Design

- A System Cannot Be Said to Be Secure if the Design of the Actual System Cannot Be Articulated
- Match Diligence to Risk
- Apply Stringent Quality Criteria to High-Risk Components
- Minimize the Number, Size, and Complexity of High-Risk Components
- Reliable Components Should Adhere to a Carefully Crafted or Applied Design Pattern
- Employ Security Design Patterns That Are Verifiable
- Minimize the Accessibility of Sensitive Components
- Prefer Highly Reliable Components for Sensitive Functions

### Realms of Trust

- Authorization Should Be Based on Trustworthiness and Appropriateness
- Distinguish Between Trust and Authorization
- Consider Hierarchical Realms and Inherited Security Policies When Possible
- Provide for Decentralized Administration of Application Security Policy
- Clearly Define the Meaning of Hierarchical Relationships

### Compartmentalization and Classification

- Segment Data and Resources According to Risk or Sensitivity
- Compartmentalize Roles
- Enforce Separation of Duties
- Roles That Grant Access Rights Must Have Those Same Rights
- Ensure That Traceability Functions Such as Logging and Monitoring Are Independent of Other Application Operation Functions
- If Two Roles Are Required to Be Independent, Their Communication Channels Should Also Be Independent
- For Sensitive Operations, Enforce One Action at a Time
- Delay the Effect of Sensitive Operations
- Implement Single Sign-On Sparingly
- Prefer to Employ Different Credentials for Each Role
- Create Unique Roles or User Identities for Each Server Application, with Only the Permissions That Are Required
- Provide a Server's Execution Role or User Identity with Exclusive Access to the Resources That the Server Owns
- Compartmentalize Server Connections

### Transport and Storage of Secrets

- Transport Sensitive Data Securely
- Do Not Store Unsecured Data in a Secure Location
- Never Store Secrets Insecurely
- Employ Secure Credential Practices
- Periodically Replace Secrets
- Distribute Credentials Securely
- Employ Multiple Independent Factors or Channels
- Treat Alternative User Verification Methods as Forms of Authentication

### Design Considerations for Secure Operation

- Deploy Securely
- Provide for Secure Restoration
- Do Not Thwart Infrastructure Security Mechanisms
- Do Not Secretly "Tunnel"
- Run Securely
- Log Security-Related Events
- Embed Intrusion Detection at Multiple Points
- Multiple Independent Means Should Be Used to Contact Responders
- Allocate Resource Capacity Based On Trust
- When Any System Is Attacked, All Other Systems Should Temporarily Convert to a Highly Safe Reduced Access Mode
- Provide Convenient API for Intrusion Response
- Provide Fine-Grained Monitoring
- Make Sure the User Is Sufficiently Informed
- Inform the User of Activity
- Educate Users of the Means by Which They Will Be Contacted in an Emergency
- Reliably Inform the User of Status
- Do Not Design Security-Related Features That Are Onerous to Use
- Do Not Assume That Users Will Be Able to Institute Complex Processes or Controls
- Require Security Features to Operate Efficiently and Quickly
- Carefully Consider the Operational Effects of Inconsistency
- Any Unavoidable Deviations from Security-Sensitive Procedures Should Be Fully Traceable
- Maintain Awareness of Vulnerabilities in Components

### Access Control Containers

- Ensure That an Authorization Mechanism Cannot Be Bypassed
- Employ Moderate Diversity
- Authorization Rules Should Be Auditable
- Avoid Duplication of Access Rules
- Use Multi-Tier ("Deep") Authorization Perimeters
- Fine-Grained Authorization (Least Privilege) at Each Perimeter
- Distinguish Between Workflow and Core Business Processes
- Distinguish Between Client Workflow and Business Workflow

- Treat Administrative Processes as Business Processes
- Model Authorization in Terms of Permissions
- Permissions for a Resource Should Be Defined to Be Orthogonal
- Distinguish Between Policy and Resource Authorization Models
- Factor Out Role Parameters as Configurable Policy
- Distinguish Between Roles and Conditions
- Default to the Most Secure (Least Permission) Level
- If a Critical Authorization or Trust Service Is Unreachable, Do Not Use a Cached Value
- Deploy with a Secure Default Configuration
- Define Authorization Models Alongside the Resources They Protect
- Tightly Associate Resource Authorization Rules with the Definition of the Input Context They Require
- Do Not Expose Internal Objects or Attributes
- Do Not Expose Internal Objects Via Collections
- Employ a Collaboration Pattern for Intra-Layer Communication
- Employ a Strongly Typed Language and Design Approach for the Interface
- Securely Transform and Compare Values
- Expose as Few Pathways and Interfaces as Possible
- Design Secure Resource Interfaces to Be "Atomic"
- Careful Design-Time Consideration of Error Conditions
- Provide No Denial Information That Could Be Useful to an Attacker
- Filter Sensitive Data from Exceptions
- Securely Log Sensitive Data Rather Than Discard It
- Ensure That Logs Are Secure, Unalterable, and Reliable
- Do Not Trust Context Data Beyond the Scope of the Context to Which the Data Applies
- Do Not Trust an Unauthenticated Context
- Identify and Assess All Context Bindings
- A Trusted Context Should Include a Proven Identity
- A Trusted Context Should Always Include the Code Context
- Validate Anything Received from Untrusted Sources
- Set Session and Credential Scope Appropriately
- Provide for Timely Revocation of Privileges
- Propagate Context Accurately or Pessimistically
- Require Every Execution Context to Have Permission
- Define Privileged Context Boundaries
- Tightly Encapsulate Privileged Operations
- Package Resources of Equal Trust and Equal Capability Requirements Together
- Do Not Intermix Business Logic with Infrastructure or Other Non-Business Logic, Unless the Separation Is Simple and Clear
- Application-Specific Dependencies Should Flow Only Toward the Business Model
- Layers Should Not Be Bypassed
- Objects Passed to Lower Layers Should Be Expressed in the Types Defined in Those Layers
- Objects Returned to Higher Layers May Be Translated by Each Layer into Types That Are Meaningful in That Layer
- A Software Layer Should Define an "Interface" for Use by Upper Layers Only
- Transfer of Information from One Domain to Another Should Always Be Through an Interface That Considers the Trustworthiness and Authorization of Each Domain
- Specify Closure
- Implement Closure
- Clearly Delineate Which Components Share Internal Dependencies
- Tightly Control Access to Resource References
- Tightly Scope All Resources
- Securely Connect to All Resources
- Prefer to Initiate Connections from a Point of Trust
- Do Not Make Blind Callbacks
- Prefer Stateful Protocols
- Rely on Secure Scoping Mechanisms
- Securely Identify All Resources
- Canonicalize All Security-Sensitive Names
- Prefer Static Resource Paths
- Secure Non-Run-Time Resources
- Secure Configuration Data
- Secure Administrative Resources
- Secure Native Resources
- Encapsulate Native Resource Access
- Secure All Caches
- Erase Sensitive Data Immediately After Use
- Dispose of Sensitive Data Securely
- Provide Secure Locations for Failure Artifacts
- Authorization Should Be a Precondition for Execution of Any and All Business Logic Within a Function or Method.

### Compositional Integrity

- Clearly Define the Roles and Responsibilities of Each Component
- Safely Validate All Inputs to Each Method
- Use Specific Argument Types Instead of General-Purpose Types
- Methods That Update State That Is Accessible Beyond the Scope of the Method Call Should Be Clearly Identified As Such
- Validate External Components

### Concurrency

- Concurrent Execution Should Not Cause Application Programs to Malfunction
- Coordinate Error Response Across All Independent Paths of Control

### Transactional Integrity

- Use Maximum Transaction Isolation by Default; Reduce Isolation on a Case-by-Case Basis, Only When Analysis Proves That the Transaction Will Still Be Sound
- Maintain Only One Copy of Each Datum Within a Transaction
- Write Through Persistent Changes Before Data Might Be Re-Read
- Document Side Effects Within a Transaction
- There Should Be a Clear and Consistent Transaction Boundary
- Employ a Timer to Abort Any Resource Call That Could Potentially "Hang"
- Build a Regression Test Suite for the Application's Transactions
- Test That the Application Works Under Concurrent Attempts to Access the Same Transactions and Data
- Test All Conceivable Failure Scenarios

### Manageability

- A Manageable Application Should Verify the Validity of Its Configuration
- A Manageable Application Should Expose Its Configuration Parameters Ranges
- Installation Process Should Be Safe
- A Manageable Application Should Be Designed According to Composability Principles
- A Manageable Application Should Generate Credible and Monitorable "Application-Is-Operational" Events
- A Manageable Application Should Generate Monitorable Progress Events at the Beginning and End of and at Frequent Intervals Within Each Significant Processing Step
- Monitor Entry and Exit From and Progress Within Time-Consuming Methods
- Report Resource Usage

### Maintainability

- Maintainable Software Should Document Why Each Action Is Performed
- Maintainable Software Should Document Functional Behavior and Requirements as Well as Non-Functional Behavior and Requirements
- Document All Exception Conditions That Can Occur
- Apply Stringent Quality Criteria to Critical Business Components
- Avoid Duplication
- Critical Business Rules Should Be Auditable
- Minimize the Number, Size, and Complexity of Critical Business Rules
- An Application That Has No Verifiable Design Must Be Tested Much More Thoroughly in Order to Provide the Same Level of Reliability
- Do Not Weaken Any Assumptions Made by a Superclass
- Testing of Maintainable Software Should Be Automated from the Beginning
- Maintainable Software Should Employ Stubs for Infrastructure That Is Not Easily Deployed in the Development Environment

### Failure Response Design

- Indicate Failure Scope
- Use Checked Exceptions for All Lifecycle Errors
- Safely Transform and Compare Types and Values
- Throw Specific Exceptions—Not Generic Ones
- Log Only at Predetermined Layer interfaces
- Time Critical Thread Actions
- Error Handlers Should Themselves Completely Trap All Errors That Might Prevent Them from Propagating Contextual Information
- Provide Context to Correlate User Actions with Server Actions
- Never Discard Exception Information
- Provide Problem Transparency
- Map Errors to Problematic Input
- Provide Management "Aspects" to Exceptions
- Alert Only Those Who Need to Know
- Provide User Status and Actionability
- Build with Debug Information, and Deploy Initially with Debug Information—Subject to Security Concerns
- Display Actual Configuration
- Actively Monitor Resource Levels to Prevent Problems Before They Occur
- An Application Console Should Monitor and Interpret Application Events in Order to Assess the Progress of Application Tasks

### Methodological Considerations

- Ensure That Someone Competent Is Responsible for the Application Architecture
- Ensure That All Implied or Assumed Requirements Are Identified
- Ensure That Critical Business Rules Are Auditable
- Ensure That the Quality-Critical Codebase Is Identified
- Ensure That an Architecture Exists That Explains How the Application Will Meet the Assurance Requirements
- Ensure the Quality and Faithfulness of Implementation with Respect to the Design
- Ensure That Architectural Concepts and Decisions Are Documented and Disseminated
- Ensure That Design Is Reviewed by Different Areas of Expertise
- Ensure That There Exists Full Configuration Management of All Artifacts
- Do Not Use Production Data as a Primary Test Suite