



CANONICAL LIVING OVERVIEW GUIDE

Application, API, and Web Security Engineering

An application-security guide covering secure architecture, browser and web controls, APIs, authentication, authorization, sessions, input and output, cryptography, data, business logic, abuse resistance, testing, verification, deployment, monitoring, and incident readiness.

PERMANENT PUBLICATION IDENTITY

Application, API, and Web Security Engineering

Document ID
MYTHOS-FG-SEC-0013

Version
1.0.0-candidate

Status
Candidate Focused Field Guide

Review date
2026-10-26

Publication position
This is a living field guide. It is designed for frequent revision while retaining a stable permanent identity. Candidate status means the content is ready for structured use and review but is not represented as an external certification or authority publication.

LEVEL L1-L4 Foundational through frontier	CLASS FIELD GUIDE Practical and educational	CADENCE QUARTERLY Interim updates as needed	EVIDENCE SOURCE-BOUND Limitations remain visible
--	--	--	---

Reader orientation

Dimension	Engineering position
Primary domain	MYTHOS-SEC - Cybersecurity, Security Engineering & Cryptography
Secondary domain	MYTHOS-SWE; MYTHOS-DAT; MYTHOS-CLD
Audience	Software, application security, API, web, cloud, platform, product, test, and security engineers.
Prerequisites	Software architecture, HTTP, APIs, identity, databases, cryptography, testing, and deployment fundamentals.
Publisher / owner	Mythos Systems / Aaron Stovall
Public classification	Public technical guidance

Expected outcomes

- Translate application risks into architecture, requirements, code, tests, and operational controls.
- Implement authentication, authorization, session, API, browser, data, and cryptographic controls correctly.
- Design for abuse resistance, multitenancy, availability, privacy, and secure failure.
- Verify applications using ASVS-aligned requirements, code review, automated tests, and adversarial testing.
- Operate applications with telemetry, secrets, patching, incident response, and continuous assurance.

How to use this guide

Read the guide as a progressive system rather than a disconnected encyclopedia. Foundational material establishes the mental model; practitioner sections convert it into repeatable implementation and operations; advanced sections expose architecture, scale, security, and failure behavior; frontier sections describe technologies whose evidence or maturity is still evolving.

Suggested reading path

New practitioners should follow the chapters in order and complete the laboratories. Experienced engineers may begin with the architecture, failure, validation, or frontier sections, then use the related-document map to deepen a specific topic.

Learning and assurance model

Level	Reader outcome
L1 - Foundational	Terminology, first principles, component roles, packet or state flow, and simple working examples.
L2 - Practitioner	Implementation, configuration, automation, testing, troubleshooting, and operational evidence.
L3 - Advanced	Architecture, scale, concurrency, resilience, threat models, performance, and complex migrations.
L4 - Frontier	Emerging systems, research directions, technical uncertainty, readiness gates, and adoption signals.

Contents

How to use this guide	3
Contents	3
Chapter 01 - Secure application architecture and requirements	5
Chapter 01 laboratory and review	10
Chapter 02 - Authentication, sessions, and account lifecycle	11
Chapter 02 laboratory and review	16
Chapter 03 - Authorization and business logic	17
Chapter 03 laboratory and review	22
Chapter 04 - Input, output, injection, and interpreter safety	23
Chapter 04 laboratory and review	28
Chapter 05 - API, HTTP, browser, and client security	29
Chapter 05 laboratory and review	34

Chapter 06 - Cryptography, secrets, and data protection	35
Chapter 06 laboratory and review	40
Chapter 07 - Verification, testing, and secure development	41
Chapter 07 laboratory and review	46
Chapter 08 - Deployment, operations, and incident readiness	47
Chapter 08 laboratory and review	52
Integrated learning roadmap	53
Source authority register	54
Limitations, freshness, and revision control	55

CHAPTER 01

Secure application architecture and requirements

Build security into system boundaries, data flows, and acceptance criteria.

Chapter operating position

Build security into system boundaries, data flows, and acceptance criteria.



1.1 Assets, trust boundaries, and threat model

Assets, trust boundaries, and threat model must be implemented as an observable engineering mechanism, not a product label or maturity claim. Identify users, services, data, identities, devices, administrators, third parties, clients, and attacker paths. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for assets, trust boundaries, and threat model.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating assets, trust boundaries, and threat model as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for assets, trust boundaries, and threat model.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

1.2 Security requirements and ASVS levels

Security requirements and ASVS levels must be implemented as an observable engineering mechanism, not a product label or maturity claim. Select verification depth according to risk and translate controls into testable acceptance criteria. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for security requirements and asvs levels.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Implementation sketch

```
authorization:
  subject: user
  tenant: derived-from-verified-session
  object: invoice
  action: read
  checks:
    - session-active
    - subject-tenant-equals-object-tenant
    - relationship-permits-action
    - object-state-allows-action
  deny_by_default: true
  audit:
    include: [subject, tenant, object, action, decision, policy-version]
tests:
  - horizontal-access-denied
  - cross-tenant-access-denied
  - stale-session-denied
```

Failure patterns

Treating security requirements and asvs levels as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for security requirements and asvs levels.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.



1.3 Secure design and failure semantics

Secure design and failure semantics must be implemented as an observable engineering mechanism, not a product label or maturity claim. Use least privilege, isolation, minimization, safe defaults, fail secure, explicit errors, and recovery. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric

paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define authority, ownership, scope, dependencies, limits, and acceptance criteria for secure design and failure semantics.
- Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.
- Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.
- Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.
- Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.
- Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

- Treating secure design and failure semantics as a product feature instead of an end-to-end engineering system.
- Relying on intended policy, rule code, or controller state without proving applied and operational behavior.
- Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.
- Changing several variables before preserving evidence and stating a falsifiable hypothesis.
- Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.
- Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for secure design and failure semantics.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

1.4 Multitenancy and administrative planes

Multitenancy and administrative planes must be implemented as an observable engineering mechanism, not a product label or maturity claim. Separate tenants, data, keys, caches, jobs, logs, support, operations, and privileged workflows. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for multitenancy and administrative planes.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating multitenancy and administrative planes as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for multitenancy and administrative planes.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

Chapter 01 laboratory and review

Practical laboratory

Create a threat model and ASVS-aligned requirement set for a multitenant web and API service, including administrative functions.

Laboratory evidence package

Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.

Versioned configuration or code with explanatory comments and reproducible setup instructions.

Nominal-path validation plus at least two injected failure or boundary conditions.

Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.

A concise decision record covering findings, limitations, residual risks, and next actions.

Frontier extension

Extend application architecture to agentic workflows, model APIs, confidential services, and quantum-safe trust.

Review gate

Advance only when another engineer can reproduce the result, explain the architecture, identify the failure boundary, and distinguish measured evidence from assumptions or projections.

CHAPTER 02

Authentication, sessions, and account lifecycle

Protect identity from enrollment through recovery and termination.

Chapter operating position

Protect identity from enrollment through recovery and termination.

2.1 Registration and proofing

Registration and proofing must be implemented as an observable engineering mechanism, not a product label or maturity claim. Validate identity, email or phone, invitations, fraud, duplicates, consent, privacy, and activation. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for registration and proofing.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating registration and proofing as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for registration and proofing.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

2.2 Authenticator and MFA design

Authenticator and MFA design must be implemented as an observable engineering mechanism, not a product label or maturity claim. Use passkeys or phishing-resistant options, secure password handling, rate limits, enrollment, and recovery. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for authenticator and mfa design.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Implementation sketch

```
authorization:
  subject: user
  tenant: derived-from-verified-session
  object: invoice
  action: read
  checks:
    - session-active
    - subject-tenant-equals-object-tenant
    - relationship-permits-action
    - object-state-allows-action
  deny_by_default: true
  audit:
    include: [subject, tenant, object, action, decision, policy-version]
tests:
  - horizontal-access-denied
  - cross-tenant-access-denied
  - stale-session-denied
```

Failure patterns

Treating authenticator and mfa design as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for authenticator and mfa design.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.



2.3 Session management

Session management must be implemented as an observable engineering mechanism, not a product label or maturity claim. Control tokens, cookies, rotation, binding, idle and absolute lifetime, concurrent sessions, revocation, and logout. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric

paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define authority, ownership, scope, dependencies, limits, and acceptance criteria for session management.
- Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.
- Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.
- Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.
- Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.
- Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

- Treating session management as a product feature instead of an end-to-end engineering system.
- Relying on intended policy, rule code, or controller state without proving applied and operational behavior.
- Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.
- Changing several variables before preserving evidence and stating a falsifiable hypothesis.
- Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.
- Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for session management.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

2.4 Account recovery and lifecycle

Account recovery and lifecycle must be implemented as an observable engineering mechanism, not a product label or maturity claim. Protect reset, support, email changes, factor replacement, suspension, deletion, and reactivation. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for account recovery and lifecycle.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating account recovery and lifecycle as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for account recovery and lifecycle.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

Chapter 02 laboratory and review

Practical laboratory

Implement or assess registration, login, MFA, session, logout, and recovery; test fixation, replay, theft, enumeration, and support abuse.

Laboratory evidence package

Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.

Versioned configuration or code with explanatory comments and reproducible setup instructions.

Nominal-path validation plus at least two injected failure or boundary conditions.

Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.

A concise decision record covering findings, limitations, residual risks, and next actions.

Frontier extension

Explore device-bound sessions, continuous authentication, privacy-preserving fraud controls, and post-quantum credentials.

Review gate

Advance only when another engineer can reproduce the result, explain the architecture, identify the failure boundary, and distinguish measured evidence from assumptions or projections.

CHAPTER 03

Authorization and business logic

Enforce permission on every object, action, field, tenant, and workflow transition.

Chapter operating position

Enforce permission on every object, action, field, tenant, and workflow transition.

3.1 Object and function authorization

Object and function authorization must be implemented as an observable engineering mechanism, not a product label or maturity claim. Verify subject, tenant, object, operation, ownership, role, relationship, and environmental constraints server side. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for object and function authorization.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating object and function authorization as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for object and function authorization.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

3.2 Workflow and state integrity

Workflow and state integrity must be implemented as an observable engineering mechanism, not a product label or maturity claim. Enforce valid transitions, prerequisites, approvals, idempotency, concurrency, limits, and rollback. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for workflow and state integrity.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Implementation sketch

```
authorization:
  subject: user
  tenant: derived-from-verified-session
  object: invoice
  action: read
  checks:
    - session-active
    - subject-tenant-equals-object-tenant
    - relationship-permits-action
    - object-state-allows-action
  deny_by_default: true
  audit:
    include: [subject, tenant, object, action, decision, policy-version]
tests:
  - horizontal-access-denied
  - cross-tenant-access-denied
  - stale-session-denied
```

Failure patterns

Treating workflow and state integrity as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for workflow and state integrity.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.



3.3 Mass assignment and field controls

Mass assignment and field controls must be implemented as an observable engineering mechanism, not a product label or maturity claim. Use explicit schemas, allowlists, immutable fields, derived values, and server-controlled security attributes. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric

paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define authority, ownership, scope, dependencies, limits, and acceptance criteria for mass assignment and field controls.
- Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.
- Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.
- Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.
- Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.
- Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

- Treating mass assignment and field controls as a product feature instead of an end-to-end engineering system.
- Relying on intended policy, rule code, or controller state without proving applied and operational behavior.
- Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.
- Changing several variables before preserving evidence and stating a falsifiable hypothesis.
- Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.
- Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for mass assignment and field controls.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

3.4 Business abuse and fraud resistance

Business abuse and fraud resistance must be implemented as an observable engineering mechanism, not a product label or maturity claim. Model automation, scraping, promotion abuse, inventory, payments, account takeover, and resource exhaustion. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for business abuse and fraud resistance.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating business abuse and fraud resistance as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for business abuse and fraud resistance.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

Chapter 03 laboratory and review

Practical laboratory

Test an API for object, function, property, tenant, and workflow authorization with both horizontal and vertical abuse cases.

Laboratory evidence package

Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.

Versioned configuration or code with explanatory comments and reproducible setup instructions.

Nominal-path validation plus at least two injected failure or boundary conditions.

Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.

A concise decision record covering findings, limitations, residual risks, and next actions.

Frontier extension

Explore relationship-based policy, cryptographic capabilities, formal workflow verification, and agent delegation.

Review gate

Advance only when another engineer can reproduce the result, explain the architecture, identify the failure boundary, and distinguish measured evidence from assumptions or projections.

CHAPTER 04

Input, output, injection, and interpreter safety

Control data crossing parser, query, template, command, and rendering boundaries.

Chapter operating position

Control data crossing parser, query, template, command, and rendering boundaries.

4.1 Validation and canonicalization

Validation and canonicalization must be implemented as an observable engineering mechanism, not a product label or maturity claim. Use typed schemas, length, ranges, formats, encoding, normalization, parser consistency, and rejection. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for validation and canonicalization.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating validation and canonicalization as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for validation and canonicalization.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

4.2 Injection prevention

Injection prevention must be implemented as an observable engineering mechanism, not a product label or maturity claim. Use parameterized queries, safe APIs, escaping by context, command isolation, and least-privilege interpreters. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for injection prevention.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Implementation sketch

```
authorization:
  subject: user
  tenant: derived-from-verified-session
  object: invoice
  action: read
  checks:
    - session-active
    - subject-tenant-equals-object-tenant
    - relationship-permits-action
    - object-state-allows-action
  deny_by_default: true
  audit:
    include: [subject, tenant, object, action, decision, policy-version]
tests:
  - horizontal-access-denied
  - cross-tenant-access-denied
  - stale-session-denied
```

Failure patterns

Treating injection prevention as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for injection prevention.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

4.3 Serialization and deserialization

Serialization and deserialization must be implemented as an observable engineering mechanism, not a product label or maturity claim. Control types, polymorphism, object creation, references, signatures, size, recursion, and untrusted formats. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric

paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define authority, ownership, scope, dependencies, limits, and acceptance criteria for serialization and deserialization.
- Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.
- Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.
- Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.
- Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.
- Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

- Treating serialization and deserialization as a product feature instead of an end-to-end engineering system.
- Relying on intended policy, rule code, or controller state without proving applied and operational behavior.
- Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.
- Changing several variables before preserving evidence and stating a falsifiable hypothesis.
- Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.
- Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for serialization and deserialization.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

4.4 Output encoding and browser contexts

Output encoding and browser contexts must be implemented as an observable engineering mechanism, not a product label or maturity claim. Encode for HTML, attributes, JavaScript, CSS, URL, headers, templates, and document generation. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for output encoding and browser contexts.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating output encoding and browser contexts as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for output encoding and browser contexts.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

Chapter 04 laboratory and review

Practical laboratory

Build a test matrix for SQL, command, template, LDAP, header, path, and browser injection; verify fixes with negative tests.

Laboratory evidence package

Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.

Versioned configuration or code with explanatory comments and reproducible setup instructions.

Nominal-path validation plus at least two injected failure or boundary conditions.

Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.

A concise decision record covering findings, limitations, residual risks, and next actions.

Frontier extension

Explore language-level memory safety, typed query builders, sandboxed interpreters, and proof-carrying inputs.

Review gate

Advance only when another engineer can reproduce the result, explain the architecture, identify the failure boundary, and distinguish measured evidence from assumptions or projections.

CHAPTER 05

API, HTTP, browser, and client security

Engineer the protocol and client boundary rather than relying on perimeter protection.

Chapter operating position

Engineer the protocol and client boundary rather than relying on perimeter protection.



5.1 HTTP and API contracts

HTTP and API contracts must be implemented as an observable engineering mechanism, not a product label or maturity claim. Use methods, status, content types, versioning, schemas, limits, pagination, idempotency, and error discipline. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for http and api contracts.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating http and api contracts as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for http and api contracts.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

5.2 OAuth, OIDC, and delegated access

OAuth, OIDC, and delegated access must be implemented as an observable engineering mechanism, not a product label or maturity claim. Use current best practices for redirects, PKCE, audiences, scopes, tokens, clients, and resource servers. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for oauth, oidc, and delegated access.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Implementation sketch

```
authorization:
  subject: user
  tenant: derived-from-verified-session
  object: invoice
  action: read
  checks:
    - session-active
    - subject-tenant-equals-object-tenant
    - relationship-permits-action
    - object-state-allows-action
  deny_by_default: true
  audit:
    include: [subject, tenant, object, action, decision, policy-version]
tests:
  - horizontal-access-denied
  - cross-tenant-access-denied
  - stale-session-denied
```

Failure patterns

Treating oauth, oidc, and delegated access as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for oauth, oidc, and delegated access.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

5.3 Browser security controls

Browser security controls must be implemented as an observable engineering mechanism, not a product label or maturity claim. Apply CSP, origin policy, CORS, cookie attributes, CSRF defenses, clickjacking protection, and trusted types. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric

paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for browser security controls.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating browser security controls as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for browser security controls.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

5.4 API abuse and availability

API abuse and availability must be implemented as an observable engineering mechanism, not a product label or maturity claim. Control discovery, inventory, authorization, resource consumption, automation, SSRF, third-party APIs, and unsafe consumption. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for api abuse and availability.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating api abuse and availability as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for api abuse and availability.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

Chapter 05 laboratory and review

Practical laboratory

Test an application across OAuth, CORS, CSRF, CSP, SSRF, rate limits, inventory, and third-party API failure conditions.

Laboratory evidence package

Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.

Versioned configuration or code with explanatory comments and reproducible setup instructions.

Nominal-path validation plus at least two injected failure or boundary conditions.

Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.

A concise decision record covering findings, limitations, residual risks, and next actions.

Frontier extension

Explore privacy-preserving APIs, signed requests, capability URLs, confidential clients, and AI tool-call authorization.

Review gate

Advance only when another engineer can reproduce the result, explain the architecture, identify the failure boundary, and distinguish measured evidence from assumptions or projections.

CHAPTER 06

Cryptography, secrets, and data protection

Protect data and keys through correct primitives, protocols, storage, and lifecycle.

Chapter operating position

Protect data and keys through correct primitives, protocols, storage, and lifecycle.

6.1 Transport and channel security

Transport and channel security must be implemented as an observable engineering mechanism, not a product label or maturity claim. Use TLS 1.3, certificate validation, host identity, secure redirects, HSTS, client authentication, and proxy boundaries. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for transport and channel security.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating transport and channel security as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for transport and channel security.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

6.2 Data at rest and field protection

Data at rest and field protection must be implemented as an observable engineering mechanism, not a product label or maturity claim. Choose encryption, tokenization, hashing, key hierarchy, search, backup, export, and deletion based on threat. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for data at rest and field protection.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Implementation sketch

```
authorization:
  subject: user
  tenant: derived-from-verified-session
  object: invoice
  action: read
  checks:
    - session-active
    - subject-tenant-equals-object-tenant
    - relationship-permits-action
    - object-state-allows-action
  deny_by_default: true
  audit:
    include: [subject, tenant, object, action, decision, policy-version]
tests:
  - horizontal-access-denied
  - cross-tenant-access-denied
  - stale-session-denied
```

Failure patterns

Treating data at rest and field protection as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for data at rest and field protection.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

6.3 Secrets and key lifecycle

Secrets and key lifecycle must be implemented as an observable engineering mechanism, not a product label or maturity claim. Use workload identity, secret stores, short-lived credentials, rotation, revocation, HSMs, and recovery. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric

paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for secrets and key lifecycle.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating secrets and key lifecycle as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for secrets and key lifecycle.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

6.4 Privacy and minimization

Privacy and minimization must be implemented as an observable engineering mechanism, not a product label or maturity claim. Limit collection, inference, access, retention, logs, analytics, exports, and administrative visibility. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for privacy and minimization.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating privacy and minimization as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for privacy and minimization.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

Chapter 06 laboratory and review

Practical laboratory

Threat-model and test transport, storage, secret, backup, log, and deletion controls for one sensitive data field.

Laboratory evidence package

Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.

Versioned configuration or code with explanatory comments and reproducible setup instructions.

Nominal-path validation plus at least two injected failure or boundary conditions.

Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.

A concise decision record covering findings, limitations, residual risks, and next actions.

Frontier extension

Explore confidential computing, FHE, ZKP, and post-quantum protocols for application data.

Review gate

Advance only when another engineer can reproduce the result, explain the architecture, identify the failure boundary, and distinguish measured evidence from assumptions or projections.

CHAPTER 07

Verification, testing, and secure development

Create layered assurance from source through runtime.

Chapter operating position

Create layered assurance from source through runtime.



7.1 Code review and static analysis

Code review and static analysis must be implemented as an observable engineering mechanism, not a product label or maturity claim. Review trust boundaries, authorization, input, errors, crypto, secrets, dependencies, concurrency, and dangerous APIs. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for code review and static analysis.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating code review and static analysis as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for code review and static analysis.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

7.2 Dynamic and interactive testing

Dynamic and interactive testing must be implemented as an observable engineering mechanism, not a product label or maturity claim. Use DAST, IAST, fuzzing, property tests, contract tests, abuse tests, and authenticated scanning. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for dynamic and interactive testing.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Implementation sketch

```
authorization:
  subject: user
  tenant: derived-from-verified-session
  object: invoice
  action: read
  checks:
    - session-active
    - subject-tenant-equals-object-tenant
    - relationship-permits-action
    - object-state-allows-action
  deny_by_default: true
  audit:
    include: [subject, tenant, object, action, decision, policy-version]
tests:
  - horizontal-access-denied
  - cross-tenant-access-denied
  - stale-session-denied
```

Failure patterns

Treating dynamic and interactive testing as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for dynamic and interactive testing.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

7.3 WSTG and adversarial testing

WSTG and adversarial testing must be implemented as an observable engineering mechanism, not a product label or maturity claim. Plan information gathering, configuration, identity, authorization, session, validation, logic, client, and API tests. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric

paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define authority, ownership, scope, dependencies, limits, and acceptance criteria for wstg and adversarial testing.
- Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.
- Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.
- Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.
- Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.
- Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

- Treating wstg and adversarial testing as a product feature instead of an end-to-end engineering system.
- Relying on intended policy, rule code, or controller state without proving applied and operational behavior.
- Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.
- Changing several variables before preserving evidence and stating a falsifiable hypothesis.
- Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.
- Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for wstg and adversarial testing.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

7.4 Release gates and evidence

Release gates and evidence must be implemented as an observable engineering mechanism, not a product label or maturity claim. Require test results, exceptions, threat-model updates, SBOM, provenance, vulnerabilities, approvals, and rollback. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for release gates and evidence.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating release gates and evidence as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for release gates and evidence.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

Chapter 07 laboratory and review

Practical laboratory

Create a CI verification pipeline mapped to application requirements, including unit, integration, fuzz, static, dynamic, and abuse tests.

Laboratory evidence package

Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.

Versioned configuration or code with explanatory comments and reproducible setup instructions.

Nominal-path validation plus at least two injected failure or boundary conditions.

Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.

A concise decision record covering findings, limitations, residual risks, and next actions.

Frontier extension

Explore autonomous security testing with sandboxed agents, reproducible evidence, and human release authority.

Review gate

Advance only when another engineer can reproduce the result, explain the architecture, identify the failure boundary, and distinguish measured evidence from assumptions or projections.

CHAPTER 08

Deployment, operations, and incident readiness

Maintain security after release across configuration, telemetry, vulnerabilities, and change.

Chapter operating position

Maintain security after release across configuration, telemetry, vulnerabilities, and change.

8.1 Secure configuration and environment

Secure configuration and environment must be implemented as an observable engineering mechanism, not a product label or maturity claim. Control headers, debug, errors, credentials, administrative interfaces, feature flags, cloud, containers, and infrastructure. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for secure configuration and environment.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating secure configuration and environment as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for secure configuration and environment.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

8.2 Application security telemetry

Application security telemetry must be implemented as an observable engineering mechanism, not a product label or maturity claim. Record authentication, authorization, sessions, data access, changes, errors, abuse, admin actions, and integrity events. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for application security telemetry.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Implementation sketch

```
authorization:
  subject: user
  tenant: derived-from-verified-session
  object: invoice
  action: read
  checks:
    - session-active
    - subject-tenant-equals-object-tenant
    - relationship-permits-action
    - object-state-allows-action
  deny_by_default: true
  audit:
    include: [subject, tenant, object, action, decision, policy-version]
tests:
  - horizontal-access-denied
  - cross-tenant-access-denied
  - stale-session-denied
```

Failure patterns

Treating application security telemetry as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for application security telemetry.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.



8.3 Vulnerability and dependency response

Vulnerability and dependency response must be implemented as an observable engineering mechanism, not a product label or maturity claim. Inventory components, patch, mitigate, test, deploy, verify, communicate, and handle unsupported dependencies. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric

paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for vulnerability and dependency response.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating vulnerability and dependency response as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for vulnerability and dependency response.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

8.4 Incident and recovery design

Incident and recovery design must be implemented as an observable engineering mechanism, not a product label or maturity claim. Support token and key revocation, containment, forensic evidence, data integrity, clean rebuild, and service restoration. The design should name authoritative inputs, identities, state transitions, dependencies, cryptographic or policy boundaries, limits, owners, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the request, identity, authorization, input, application logic, data, dependency, response, and telemetry chain. A configured integration, successful API response, established cryptographic session, green dashboard, or closed ticket is not sufficient proof. Intended state must be reconciled with applied state, negative tests, degraded behavior, raw evidence, recovery, and the resulting security or mission outcome.

A production implementation starts with a minimal known-good baseline and explicit invariants. Introduce one dependency or capability at a time, preserve before-state evidence, test authorized and prohibited behavior, inject realistic failures, measure convergence or recovery, and retain a tested rollback. Automation must stop safely when authority, freshness, identity, integrity, cryptographic state, or observed outcomes are ambiguous.

Troubleshooting locates the first divergence from the expected state machine or end-to-end transaction. Inspect every authority boundary and account for time, caching, eventual consistency, stale identity, incomplete collection, parser drift, asymmetric paths, encapsulation, supply-chain state, side channels, partial failure, and missing evidence. Closure requires causal explanation, reproducibility, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define authority, ownership, scope, dependencies, limits, and acceptance criteria for incident and recovery design.

Model expected state, trust boundaries, data flow, identity, and failure behavior before implementation.

Validate intended configuration, active state, applied enforcement, raw evidence, and end-to-end outcome independently.

Test normal, prohibited, degraded, failed, scale, recovery, and rollback conditions.

Preserve policy, source, version, identity, telemetry, artifact, packet, log, time, and decision evidence.

Automate through typed contracts, least privilege, deterministic gates, canaries, bounded authority, and reversible changes.

Failure patterns

Treating incident and recovery design as a product feature instead of an end-to-end engineering system.

Relying on intended policy, rule code, or controller state without proving applied and operational behavior.

Ignoring data quality, identity, time, cryptographic lifecycle, supplier state, or partial failure.

Changing several variables before preserving evidence and stating a falsifiable hypothesis.

Allowing automation or AI to act on stale, conflicting, unverified, or unauthorized inputs.

Declaring success after one positive result without negative tests, rollback, and recurrence monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, identities, data, algorithms, policy, configuration, ownership, and dependencies for incident and recovery design.
Observe	Active state, applied policy, raw events, telemetry, artifacts, cryptographic results, resources, and endpoint behavior.
Test	Expected behavior, prohibited behavior, malformed inputs, dependency loss, partial failure, scale, recovery, and rollback.
Measure	Availability, latency, detection quality, evidence completeness, cryptographic cost, risk, resource use, and mission outcome.
Reconcile	Intended state against observed security, identity, application, cloud, cryptographic, data, and evidence records.

Chapter 08 laboratory and review

Practical laboratory

Run an application incident involving token theft, authorization abuse, vulnerable dependency, and data export; contain and recover with evidence.

Laboratory evidence package

Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.

Versioned configuration or code with explanatory comments and reproducible setup instructions.

Nominal-path validation plus at least two injected failure or boundary conditions.

Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.

A concise decision record covering findings, limitations, residual risks, and next actions.

Frontier extension

Develop an application security digital twin combining requirements, routes, authorization, data, tests, dependencies, and runtime evidence.

Review gate

Advance only when another engineer can reproduce the result, explain the architecture, identify the failure boundary, and distinguish measured evidence from assumptions or projections.

Integrated learning roadmap

The guide is intended to be revisited. First establish fluency, then build a small implementation, operate it under failure, automate repeatable work, and finally evaluate emerging techniques against a stable baseline. Progress is demonstrated through evidence rather than reading completion alone.

Stage	Demonstrated capability
Foundation	Explain the system from first principles and trace its critical path without relying on product terminology.
Build	Implement the smallest representative system with explicit contracts, identities, telemetry, and tests.
Operate	Diagnose injected failures, recover safely, and preserve evidence of the causal chain.
Automate	Convert a proven manual workflow into bounded, idempotent, observable automation.
Architect	Design for scale, resilience, security, interoperability, and lifecycle change.
Explore	Evaluate frontier techniques using stated hypotheses, limitations, and adoption gates.

Source authority register

Source policy

Official standards, specifications, and project documentation remain with their issuing organizations. Mythos Systems provides original engineering interpretation, synthesis, implementation guidance, and relationship mapping. Source verification dates do not guarantee that an authority has not changed since review.

01. OWASP Foundation - Application Security Verification Standard 5.0

Role: Verification requirements for web application technical security controls.

Verified: 2026-07-26

Official source: <https://owasp.org/www-project-application-security-verification-standard/>

02. OWASP Foundation - OWASP Top 10:2025

Role: Current awareness baseline for critical web-application security risks.

Verified: 2026-07-26

Official source: <https://owasp.org/Top10/2025/>

03. OWASP Foundation - OWASP API Security Top 10 - 2023

Role: Current released API security risk awareness baseline.

Verified: 2026-07-26

Official source: <https://owasp.org/API-Security/editions/2023/en/0x11-t10/>

04. OWASP Foundation - Web Security Testing Guide 4.2

Role: Structured web-application security testing guidance.

Verified: 2026-07-26

Official source: <https://owasp.org/www-project-web-security-testing-guide/v42/>

05. IETF / RFC Editor - RFC 9700 - Best Current Practice for OAuth 2.0 Security

Role: Current OAuth 2.0 security best current practice.

Verified: 2026-07-26

Official source: <https://www.rfc-editor.org/info/rfc9700>

06. IETF / RFC Editor - RFC 8446 - The Transport Layer Security Protocol Version 1.3

Role: TLS 1.3 protocol specification.

Verified: 2026-07-26

Official source: <https://www.rfc-editor.org/info/rfc8446>

07. NIST - SP 800-218 - Secure Software Development Framework Version 1.1

Role: Final secure-software-development practices and tasks.

Verified: 2026-07-26

Official source: <https://csrc.nist.gov/pubs/sp/800/218/final>

08. NIST - Draft SP 800-218 Rev. 1 - Secure Software Development Framework Version 1.2

Role: Initial public draft of the next SSDF revision; used with explicit draft status.

Verified: 2026-07-26

Official source: <https://csrc.nist.gov/pubs/sp/800/218/r1/ipd>

Authority application and refresh triggers

Authority class	Required library action
Protocols and specifications	Recheck interoperability, security considerations, defaults, and migration guidance when a referenced specification changes.
Security and assurance frameworks	Update control mappings, evidence expectations, risk language, and assessment procedures after material framework revision.
Languages, runtimes, and projects	Re-run examples, tests, deprecation checks, and operational recommendations against supported releases.
Benchmarks and evaluations	Re-execute claims when workloads, methods, models, compilers, drivers, hardware, or scoring rules change.
Operational evidence	Incorporate validated incidents, failure tests, reader corrections, and measured implementation outcomes into the next revision.

Limitations, freshness, and revision control

Boundary	Statement
Publication limitation	This guide synthesizes broad engineering practice. It does not replace product documentation, legal advice, safety certification, or environment-specific design review.
Evidence limitation	Not every pattern is benchmarked in every environment. Examples illustrate engineering methods and must be validated against actual versions, workloads, hardware, policy, and failure conditions.
Freshness limitation	Vendor behavior, software libraries, AI models, security threats, and emerging standards can change quickly. Volatile claims require source revalidation before operational adoption.
Adoption limitation	Frontier content is not an implicit adoption recommendation. Adoption requires defined value, qualification gates, controlled proof, rollback, ownership, and residual-risk acceptance.
Status boundary	Candidate Focused Field Guide describes the publication, not certification of a product, implementation, engineer, or organization.

Revision record

Version	Revision statement
1.0.0-candidate	Initial candidate living-guide edition. Published 2026-07-26; scheduled review 2026-10-26.

Search and relationship metadata

Dimension	Engineering position
Primary domain	MYTHOS-SEC - Cybersecurity, Security Engineering & Cryptography
Secondary domain	MYTHOS-SWE; MYTHOS-DAT; MYTHOS-CLD
Publication class	Field Guide / Canonical Living Overview Guide
Skill levels	L1 Foundational; L2 Practitioner; L3 Advanced; L4 Frontier
Tags	application security, API security, web security, ASVS 5.0, OWASP Top 10 2025, OAuth, authorization, injection, secure coding, security testing
Canonical route	/library/field-guides/mythos-fg-sec-0013/

Living publication

Corrections, expanded laboratories, improved diagrams, authority changes, and new engineering evidence should revise this permanent publication rather than create disconnected replacements.