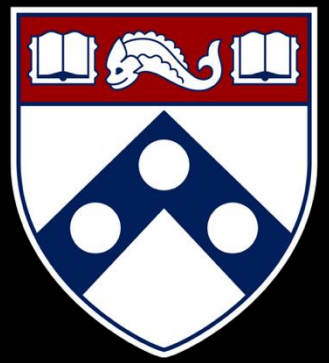
Secure Systems Engineering and Management

A Data-driven Approach

Michael Hicks



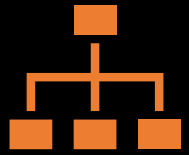
Secure Design: Principles and Controls (part 2)

Controlling Access, Limiting Damage, Detecting Attacks

UPenn CIS 7000-003
Spring 2026

From Threats to Mitigations

We **enumerate threats** against our security objectives using STRIDE.



For each threat, we **propose a response**: a control and implementation guidance.

These two lectures build the **technical vocabulary** and the **design thinking** for that.

Recall: Security Objectives

Requirement	Definition
Confidentiality	Protect information from unauthorized disclosure
Integrity	Protect information from unauthorized modification
Availability	Provide access when required
Accountability	Non-repudiation via permanent audit records

Feature	Intent
Authentication	Reliably identifying users
Authorization	Granting access based on roles and rights

Requirements expressed in terms offered by these features

Security Principles Underlying Controls

Prevention

Eliminate defects entirely

Harm reduction

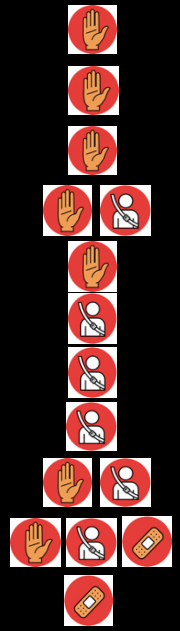
Limit damage from unknown defects

Detection & recovery

Identify attacks and **undo** damage

Secure Design Principles

- Favor simplicity
- Use fail-safe defaults
- Do not expect expert users
- Trust with reluctance
  - Small trusted computing base
  - Grant least privilege
  - Compartmentalize
  - Promote privacy
- Defend in depth
- Use community resources
- Monitor and trace



DRAFT

date October 3, 1974

THE PROTECTION OF INFORMATION IN COMPUTER SYSTEMS

by

Jerome H. Saltzer and Michael D. Schroeder

- Economy of mechanism
- Fail-safe defaults
- Psychological acceptability
- (several S&S principles)
  - Least common mechanism
  - Least privilege
  - Least privilege (again)
  - (modern addition)
- Separation of privilege
- Open design
- Compromise recording

Outline

Last time (part 1)

- **Confidentiality** (by encryption)
- **Authentication** – proving identity
- Four principles:
  - **Do not expect expert users**
  - **Fail-safe defaults**
  - **Favor simplicity**
  - **Defend in depth** (throughout)

Techniques
TLS, BLE security, AES-GCM, key management
Argon2id, MFA, IdPs, mTLS
Password managers, FIDO2
Secure defaults, break glass, graceful degradation
Minimal APIs, boring technology

Interlude

DOI:10.1145/3651621

Article development led by acmqueue
queue.acm.org

Continuous assurance at scale.

BY CHRISTOPH KERN

Developer Ecosystems for Software Safety

HOW TO DESIGN and implement information systems so they are safe and secure is a complex topic. Both high-level design principles and implementation guidance for software safety and security are well established and broadly accepted. For example, Jerome Saltzer and Michael Schroeder's seminal overview of principles of secure design was published almost 50 years ago,[10] and various community and governmental bodies have published comprehensive best practices about how to avoid common software weaknesses—for example, Common Weakness Enumeration (CWE)[a] and Open Worldwide Application Security Project (OWASP) Cheat Sheet Series.[b]

Despite these efforts, common types of software defects prevail, and many occupy top ranks of "worst vulnerabilities" lists such as the OWASP Top 10[c] or the CWE Top 25 Most Dangerous Software Weaknesses[4] for years if not decades.

Based on work at Google over the past decade on managing the risk of software defects in its wide-ranging portfolio of applications and services, the members of Google's security engineering team developed a theory about the reason for the prevalence of defects: It's simply too difficult for real-world development and operations teams to apply the available guidance comprehensively and consistently, which results in a problematic rate of new defects. Commonly used approaches to find and fix implementation defects after the fact can help (for example, code review, testing, scanning, or static and dynamic analysis such as fuzzing), but in practice they find only a fraction of these defects. Design-level defects are difficult or impractical to remediate after the fact. This leaves a problematic residual rate of defects in production systems.

We concluded that the rate at which common types of defects are introduced during design, development, and deployment is systemic—it arises from the design and structure of the *developer ecosystem*, which means the end-to-end collection of systems, tooling, and processes in which developers design, implement, and deploy software. This includes programming languages, software libraries, application frameworks, source repositories, build and deployment tooling, the production platform and its configuration surfaces, and so forth.

In short, the safety and security posture of a software application or service is substantially an *emergent property* of the developer ecosystem

a https://cwe.mitre.org/

b https://cheatsheetseries.owasp.org/

c https://owasp.org/Top10/

Why Guidance Alone Doesn’t Work

“It’s simply too difficult for real-world dev and ops teams to apply the available guidance comprehensively and consistently.”

Two reasons

1. **Developer vigilance doesn’t scale**
2. **After-the-fact tools are incomplete**

Do not expect expert users

The Ecosystem Thesis

"The safety and security of a software application or service is substantially an **emergent property of the developer ecosystem** that produced it."

To improve security, redesign the *ecosystem*, not just the guidance.

Programming languages, libraries, frameworks, build tooling, deployment infrastructure, configuration surfaces.

Strategy 1: Safe Coding

Make the ecosystem **prevent entire classes of bugs** through language design, type systems, and safe abstractions.

Google

Google Security Engineering Technical Report[1]
November 7, 2025

Safe Coding

Rigorous Modular Reasoning about Software Safety (Extended Version)

Christoph Kern
xtof@google.com

Many dangerous and persistent software vulnerabilities, including memory-safety violations and code injection, stem from a common root cause: developers unintentionally violating implicit safety preconditions when using common programming constructs. In large, complex systems, these preconditions often rely on nonlocal, whole-program invariants that are difficult for any single developer to reason about correctly and consistently. Traditional approaches such as developer education and reactive bug detection have proven insufficient to reduce these vulnerabilities to an acceptable level. Fundamentally, development environments make it too easy for well-intentioned developers to introduce subtle yet potentially catastrophic coding errors.

This article introduces *Safe Coding*, a collection of software design patterns and practices that cost-effectively provides a high degree of assurance against entire classes of such vulnerabilities. The core idea is to shift responsibility for safety from the individual developer to the programming language, libraries, and frameworks. Safe Coding achieves this by identifying *risky operations*—those with complex safety preconditions—and systematically eliminating their direct use in application code. Instead, risky operations must be encapsulated within *safe abstractions:* modules whose public APIs are safe to use by design and whose implementations take full responsibility for satisfying all internal safety preconditions. These design patterns have been successfully applied at Google, nearly eliminating classes of software vulnerabilities such as XSS (cross-site scripting) and SQL injection. Safe abstractions are also a core design principle in the Rust developer community, critical for achieving high-assurance memory safety despite the necessary presence of unsafe Rust in performance-critical and low-level systems code.

Safe Coding embodies a modular, compositional approach to building and reasoning about the safety of large, complex systems. Difficult and subtle reasoning about the safety of abstractions is localized to their implementations; the safety of risky operations within an abstraction must rely solely on assumptions supported by the abstraction's APIs and type signatures. Conversely, the composition of safe abstractions with *safe code* (i.e., code free of risky operations, which constitutes the vast majority of a program) is automatically verified by the implementation language's type checker.

While not a formal method itself, Safe Coding is grounded in principles and techniques from rigorous, formal software verification. It pragmatically adapts concepts such as function contracts and modular proofs for practical large-scale use by lifting safety preconditions into type invariants of custom data types within the chosen implementation language.

This article explores these technical and formal underpinnings, demonstrating how they enable cost-effective yet rigorous reasoning about software safety in very-large-scale industrial software development.

Software Specifications, Correctness, and Safety

In a formal sense, a program or component is *correct*, relative to a specification, when:

1. It implements all behaviors required by the specification (for example, an API service responds to requests with a specified answer or an appropriate error).
2. Every possible behavior of the program or component is permitted by the specification.

The most rigorous approach to demonstrating program correctness relies on capturing both its specification and implementation in a formal, mathematical framework such as a program logic and constructing a mathematical proof that the implementation satisfies the specification [12].

For industrial-scale software, however, developing a comprehensive formal specification—let alone formally proving

[1]This report is the extended version of an article in the Sep/Oct, 2025 issue of ACM Queue [17].

As a guide to readers of both versions, content unique to the extended version is marked with blue borders.

Strategy 1: Safe Coding

Defect Class	Guidance Approach	Safe Coding Approach
Memory corruption	“Don’t access freed memory” (SEI CERT)	Use a memory-safe language (Rust, Go, Java)
SQL injection	“Always use parameterized queries” (OWASP)	API accepts only `TrustedSqlString`, not `String`
XSS	“Always escape untrusted input” (OWASP)	Template system auto-escapes; `SafeHtml` type enforced

From “developers must remember”

to “the compiler/ framework enforces.”

The Results: Near-Zero Defect Rates

At Google, Safe Coding eliminated XSS and SQL injection as practical concerns:

- **Before:** Tens of XSS vulnerabilities per year in each large web app. **After:** Residual XSS across *all* framework-based applications in the "low single digits"
- **SQL injection:** "essentially a nonissue" in the Google internal codebase

Strategy 2: Securing Application Archetypes

Many applications share a common architecture – a **software archetype**:

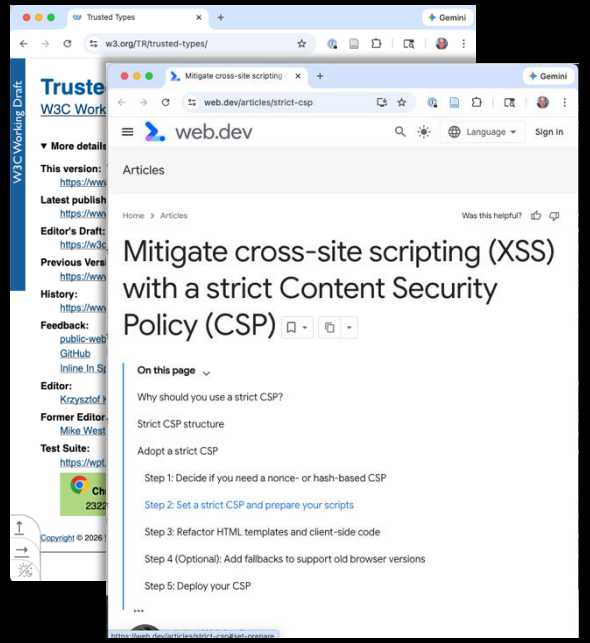
- "Web frontend + microservices + SQL database"
- "Mobile app + API server + backend services"

The threat models for each archetype substantially overlap.
Develop controls in the *framework*.

Scaling Security Through Frameworks: Evidence

The result: **Thousands of applications. Near-zero residual XSS.**

Framework-Level Action	Scale
Services newly enforcing **Trusted Types** (XSS prevention)	176
JS builds with safe-coding conformance checks on *all* transitive dependencies	1,079
Legacy DOM XSS sink assignments removed	438
Services enforcing **strict Content Security Policy**	900+
Adoption on tier-0 services (Gmail, Docs, Meet, Photos, …)	>75%



Outline

Today (part 2)

- **Authorization**
- **Integrity** and **Accountability**
- Remaining principles:
  - **Trust with Reluctance**
  - **Monitoring and Traceability**
  - **Putting it all together**
  - **Use community resources**

As we go, consider: Rather than expecting every developer to get this right, how would you build an ecosystem that makes the wrong thing impossible?

Authorization – Controlling Access

From Authentication to Authorization

Last time: **Authentication –** protecting data and proving identity.

Today: **Authorization –** once you know *who* someone is, what should they be allowed to *do*?

Primary STRIDE threat: **Elevation of Privilege**

Cedar: A Modern Authorization Policy Language

Used in **30+** Amazon services and applications
and by Cloudflare, Cloudinary, MongoDB, Salesforce, StrongDM

Part of the CLOUD NATIVE COMPUTING FOUNDATION

Open source at **https://github.com/cedar-policy**

4M
Downloads

1,400+
Stars

Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization

JOSEPH W. CUTLER*, University of Pennsylvania, USA
CRAIG DISSELKOEN, Amazon Web Services, USA
AARON ELINE, Amazon Web Services, USA
SHAOBO HE, Amazon Web Services, USA
KYLE HEADLEY*, Unaffiliated, USA
MICHAEL HICKS, Amazon Web Services, USA
KESHA HIETALA, Amazon Web Services, USA
ELEFTHERIOS IOANNIDIS*, University of Pennsylvania, USA
JOHN KASTNER, Amazon Web Services, USA
ANWAR MAMAT*, University of Maryland, USA
DARIN MCADAMS, Amazon Web Services, USA
MATT MCCUTCHEN*, Unaffiliated, USA
NEHA RUNGTA, Amazon Web Services, USA
EMINA TORLAK, Amazon Web Services, USA
ANDREW M. WELLS, Amazon Web Services, USA

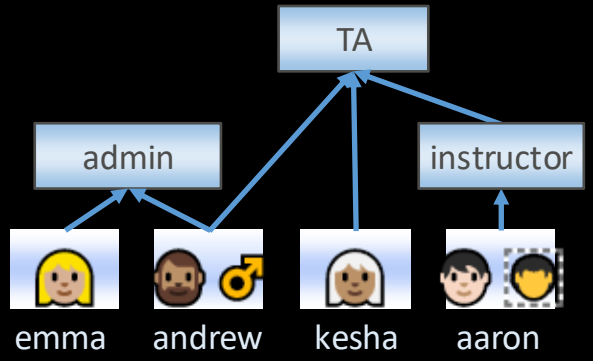
Cedar is a new authorization policy language designed to be ergonomic, fast, safe, and analyzable. Rather than embed authorization logic in an application's code, developers can write that logic as Cedar policies and delegate access decisions to Cedar's evaluation engine. Cedar's simple and intuitive syntax supports common authorization use-cases with readable policies, naturally leveraging concepts from role-based, attribute-based, and relation-based access control models. Cedar's policy structure enables access requests to be decided quickly. Cedar's policy validator leverages optional typing to help policy writers avoid mistakes, but not get in their way. Cedar's design has been finely balanced to allow for a sound and complete logical encoding, which enables precise policy analysis, e.g., to ensure that when refactoring a set of policies, the authorized permissions do not change. We have modeled Cedar in the Lean programming language, and used Lean's proof assistant to prove important properties of Cedar's design. We have implemented Cedar in Rust, and released it open-source. Comparing Cedar to two open-source languages, OpenFGA and Rego, we find (subjectively) that Cedar has equally or more readable policies, but (objectively) performs far better.

*Work carried out while at Amazon Web Services

Role-Based Access Control (RBAC)

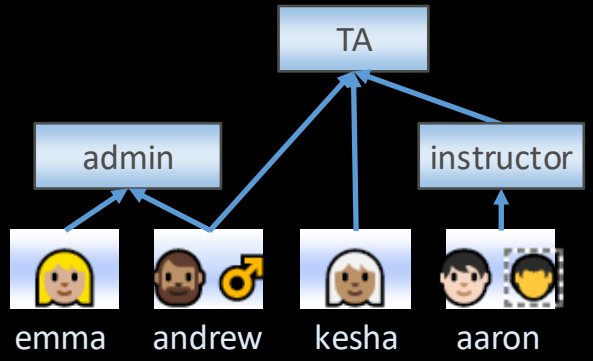
```
permit (
  principal in Role::"admin",
  action, // all actions
  resource);

permit (
  principal in Role::"TA",
  action == Action::"SetGrade",
  resource);
```

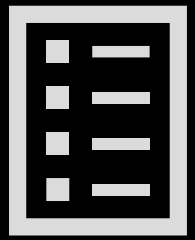
in: transitive membership



Attribute-Based Access Control (ABAC)

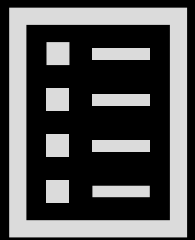
```
permit (
  principal in Role::"admin",
  action, // all actions
  resource);

permit (
  principal in Role::"TA",
  action == Action::"SetGrade",
  resource)
when {
  context.now <
  resource.course.freezeDate
};
```





Course::
"CIS7000"

instructor:
User::"aaron"
TAs: […
User::"kesha"]
freezeDate: …

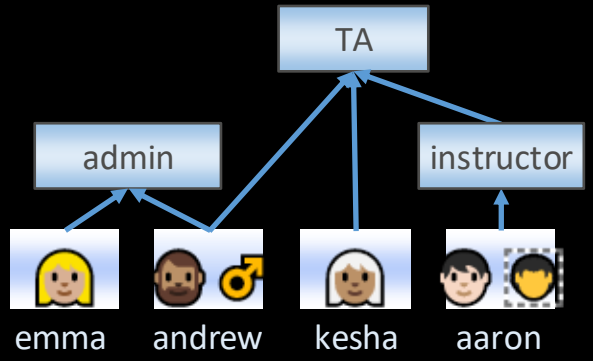


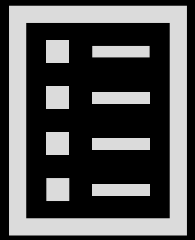
Assignment::
"Project1"

course:
Course::"CIS7000"
dueDate:
29-March-2026

Relationship-Based Access Control (ReBAC)

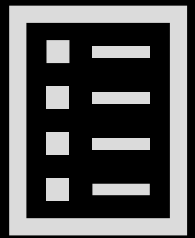
```
permit (
  principal in Role::"admin",
  action, // all actions
  resource);

permit (
  principal in Role::"TA",
  action == Action::"SetGrade",
  resource)
when {
  context.now <
  resource.course.freezeDate
}
when {
   principal in
   resource.course.Tas
|| principal ==
   resource.course.instructor
}
```





Course::
"CIS7000"

instructor:
User::"aaron"
TAs: […
User::"kesha"]
freezeDate: …



Assignment::
"Project1"

course:
Course::"CIS7000"
dueDate:
29-March-2026

Delegation and Consent

Sometimes a user must grant another access:

- Payment app: user authorizes a financial advisor to view transactions
- CMS: instructor delegates grading to a TA

Delegated access should be:

- **Scoped** – only this section/account
- **Time-bounded** – only this semester
- **Revocable** – grantor can remove it
- **Audited** – logged

Design question: What if the delegating user's own access is revoked?

→ Cascading revocation.

Complete Mediation

S&S: Every access to every object must be checked

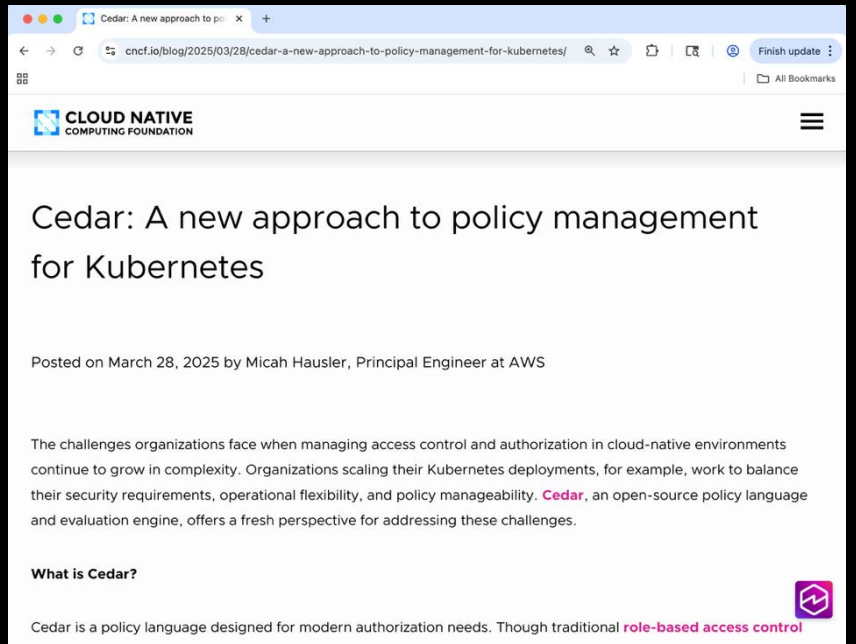
Example failure: **Insecure Direct Object Reference (IDOR)**

- CMS: `/api/students/1042/grades` – change `1042` to `1043` and see someone else's grades
- Payment app: `/api/accounts/5678/balance` – change the account ID

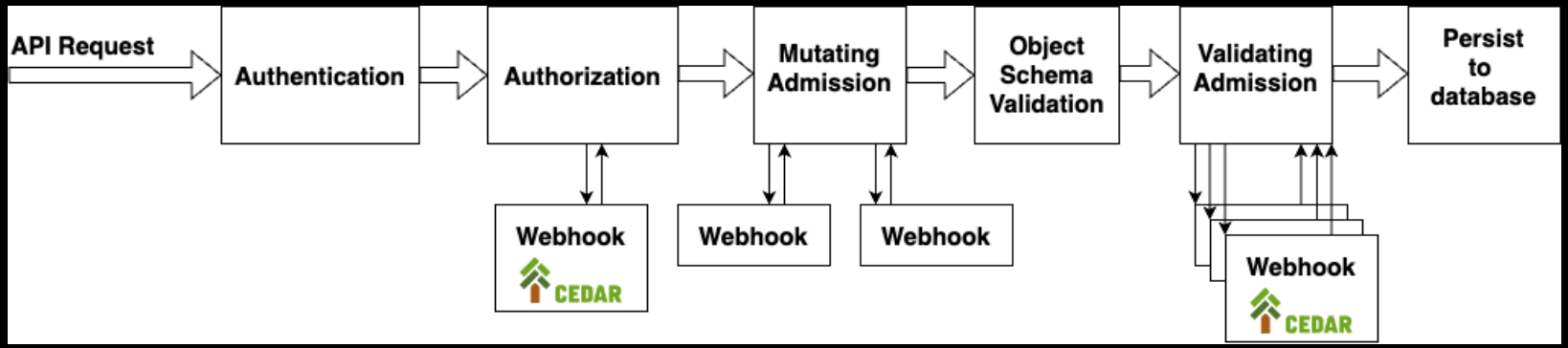
Ecosystem Assistance

Risk: With 50 route handlers, each implementing its own authorization check, real chance to get it wrong.

With platform help: One policy engine, enforced on every request by the infrastructure. Developers write declarative policies; Kubernetes enforces them.



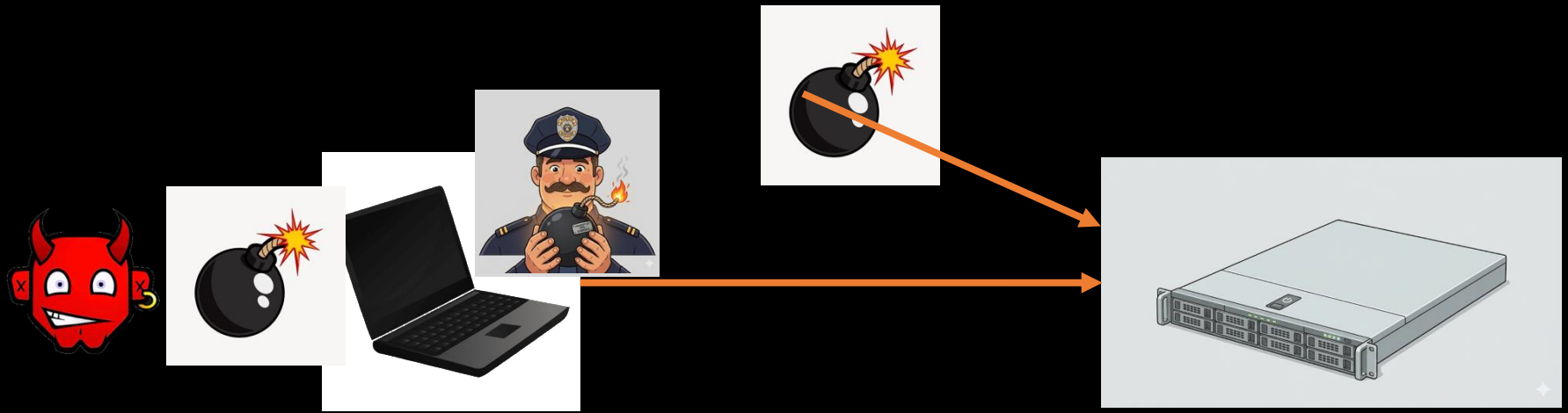
Ecosystem Assistance



https://www.cedarpolicy.com/blog/cedar-for-kubernetes

Server-Side Enforcement

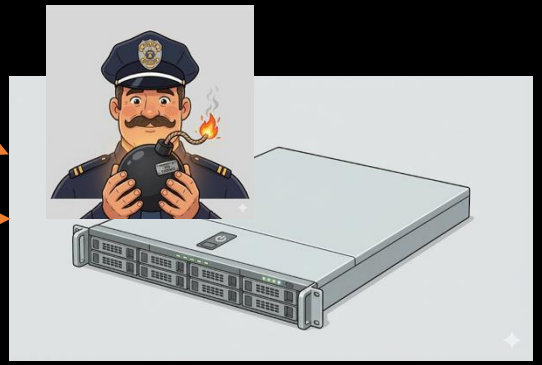
Authorization must be enforced **on the server**, not just in the client UI.



Server-Side Enforcement

Authorization must be enforced **on the server**, not just in the client UI.

Rule: the **UI is for usability**; the **server is for security**.



Fail-safe Defaults and Least Privilege in Authorization

Fail-safe defaults: The default is *no access*.

- CMS: a new instructor has access to *zero* courses until assigned
- Payment app: a new support agent can view *no* accounts until configured
- If the authorization service errors → result is **deny**, not allow

Least privilege: Each user, component, and API key gets the *minimum* permissions needed.

- CMS: notification service can send emails but can't read grades
- Payment app: push notification service has no access to bank credentials

This limits blast radius when (not if) a component is compromised

Integrity and Accountability

Data Integrity: Input Validation

Every piece of data crossing a trust boundary must be validated

Validate using **allowlists**, not denylists**:**

- **Payment app:** Transfer amount must be positive, within balance, below daily limit, at most two decimal places. Reject everything else.
- **CMS:** Allowlist of upload types (`.pdf`, `.docx`, `.zip`) rather than blocklist of dangerous ones (`.exe`, `.bat`, `.js`).

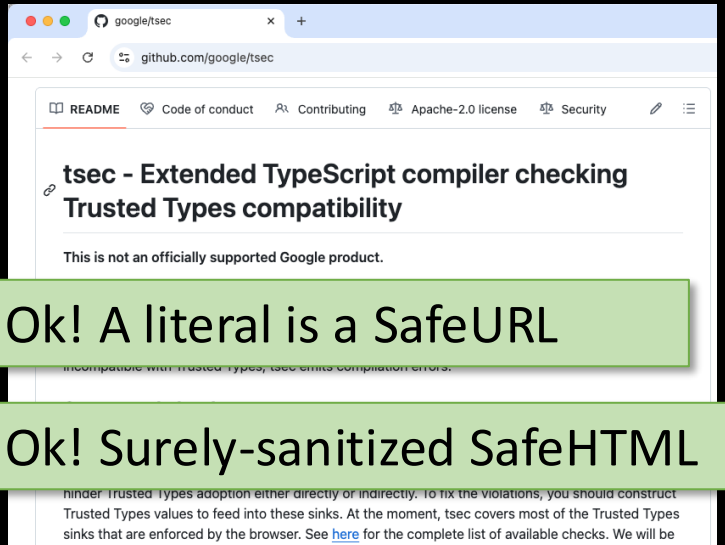
Input Validation in Google Safe Coding

```
// Navigate window to a new URL.
window.location.href = 'https://example.com';
// Append new HTML markup to the current document.
document.write(html);
```

Type error! Requires SafeURL

Type error! Requires SafeHTML

```
// Signature of a safe setter for Location#href.
// Values assigned to the sink must have the
// `SafeUrl` type.
declare function safeSetLocationHref(
  loc: Location, url: SafeUrl): void;

// Signature of a safe version of Document#write.
// Values appended must have the `SafeHtml` type.
declare function safeDocumentWrite(
  doc: Document, html: SafeHtml): void;

// Navigate window to a safe URL built from literal
safeSetLocationHref(
  window.location,
  SafeUrl.fromLiteral('https://example.com'),
);
// Append sanitized HTML markup to the current
// document.
safeDocumentWrite(document, SafeHtml.sanitize(html));
```



Ok! A literal is a SafeURL

Ok! Surely-sanitized SafeHTML

Safe Coding: Parameterized Queries

```
func (db *DB) Query(
    query string, args ...any) (*Rows, error)
```

```
q2 := "SELECT y FROM table"
q2 += " WHERE x = " + inputX
rows, err := db.Query(q2)
```

Risk! Potential SQL injection

```
type SafeDB struct {
    db *sql.DB
}

func (sdb *SafeDB) Query(
    query TrustedSqlString, args ...any) (
    *sql.Rows, error) {
    return sdb.db.Query(query.String(), args)
}
```

Trustworthy wrapper type
(e.g., via prep. statements)

Underlying DB call as usual

Communicated Message Authentication

Beyond encryption, messages need **integrity protection**: did this data arrive unmodified?

HMAC (Hash-based Message Authentication Code):

- Sender and receiver share a secret key
- Sender computes HMAC(key, message) and appends it
- Receiver recomputes and verifies
- Detects any modification

Authenticated encryption (AES-GCM): integrity is built in.

- Payment app: transfer instructions to bank API should be authenticated
- CMS: webhook callbacks from integrations should include HMAC signatures

Primary STRIDE threat: Tampering

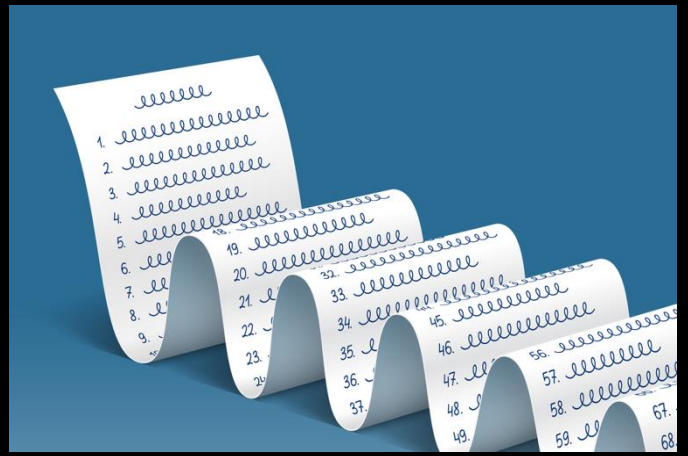
Audit Logging: The Repudiation Threat

STRIDE threat: Repudiation

- CMS: an instructor changes a grade. Later there's a complaint. Without an audit trail, it's one person's word against another's.
- Payment app: a user initiates a $500 transfer, then claims "I never authorized that."

What to log:

- Authentication attempts (success and failure)
- Authorization decisions (especially denials)
- Data modifications
- Configuration changes



Audit Log Properties

Property	Why It Matters
Append-only / write-once	Attacker who compromises system can't erase their tracks
Timestamped	Reliable clock source for ordering events
Stored separately	Compromise of app server doesn't compromise logs
Minimal sensitive data	Log "User #1234 transferred $500 to #5678" – don't log bank account numbers

Designing for observability:

- Include **correlation IDs** (request IDs, trace IDs) so events from different services for the same user action can be linked

Trust with Reluctance

The Umbrella Principle

Whole-system security depends on the secure operation of its parts. Those parts are *trusted*.

Improve security by reducing the need for trust:

- use a small trusted computing base,
- minimize transitive trust,
- reduce privilege with compartmentalization,
- promote privacy, and
- validate inputs as being well-formed

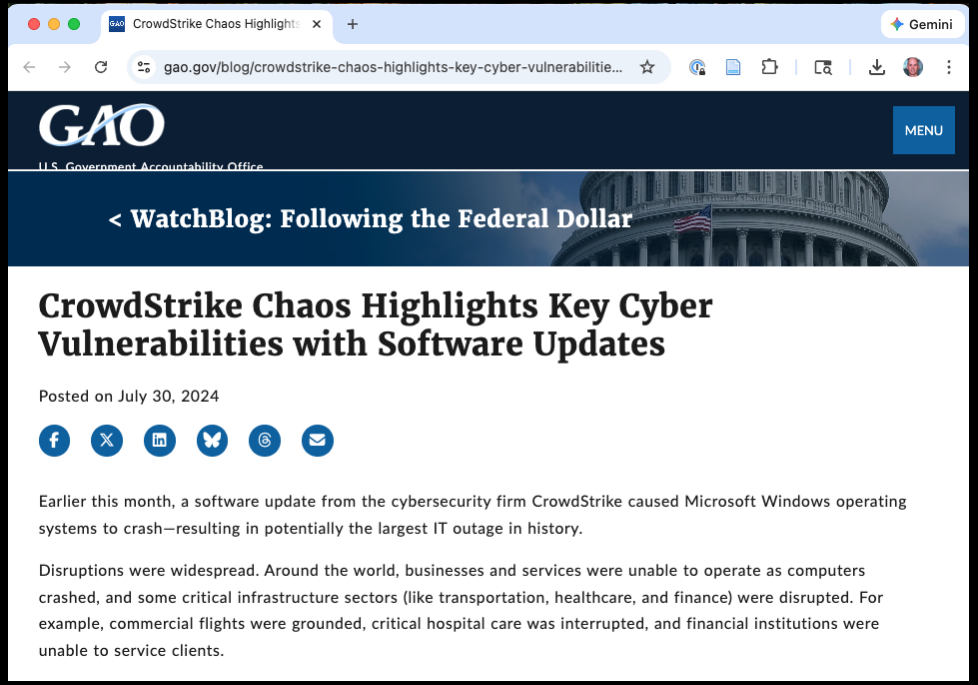
Later in course: authenticating code and data

Small Trusted Computing Base (TCB)

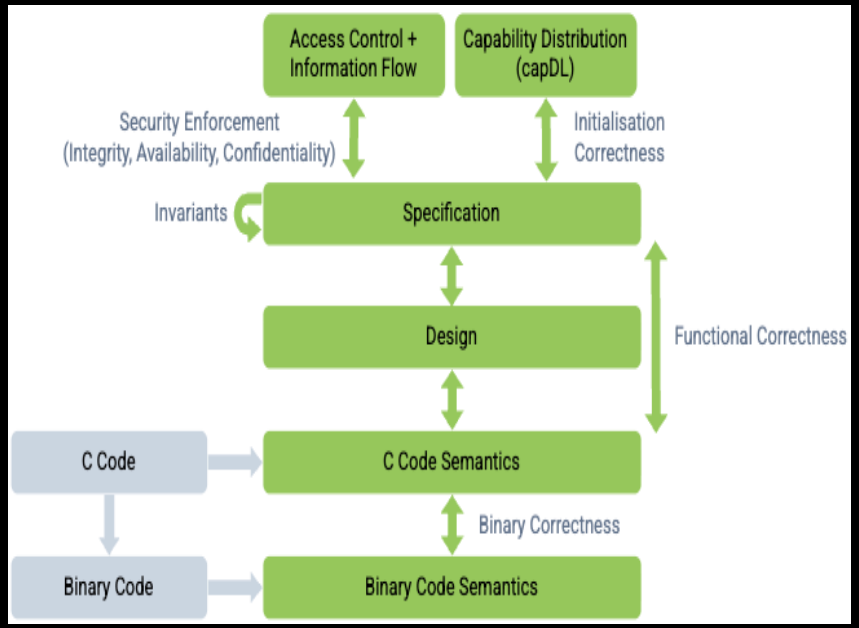
The **TCB** comprises the system components that must work correctly for security to hold.

Keep it small (and simple) to reduce susceptibility to compromise.

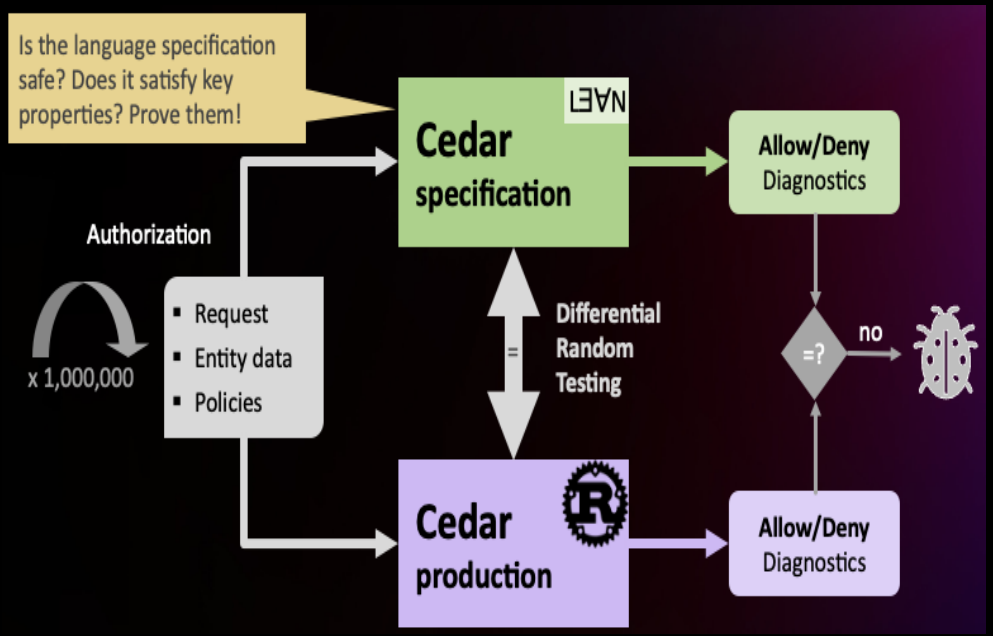
The security software paradox: Security tools (firewalls, IDSes, antivirus) are part of the TCB. But as they grow in complexity, they become vulnerable themselves.



Examples: seL4 and Cedar



seL4 security proof structure



Cedar verification-guided development

Ecosystem Lens: Frameworks as the TCB

Kern's archetype frameworks are a *deliberate TCB design*:

- **Security experts** build and maintain the framework (the TCB)
- **Application developers** build on top of it (outside the TCB)

Improving the framework improves *all* applications built on it.

But: If the framework has a bug, all applications are affected.

- A small, well-scrutinized codebase is far less likely to have bugs than thousands of ad-hoc implementations.

Config-as-Code: Safe Coding for Deployment

Framework-as-TCB includes the **production environment**: Engineers may not make direct changes to deployment; offered a limited API

The unreliable way: An engineer configures a firewall through a CLI or web UI. Changes take effect immediately, including mistakes

The safer way: A Kubernetes `NetworkPolicy`, checked into version control, and enforced by the platform

Config-as-Code: Safe Coding for Deployment

```yaml
# payment-api-network-policy.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: payment-api-policy
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: payment-api
  policyTypes:
    - Ingress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: frontend
      ports:
        - port: 443
```

Policy (like Cedar) is declarative (what, not how)

only the `frontend` can reach `payment-api`

HTTPS only

Transitive Trust

If you trust component A, and A trusts component B, then you **implicitly trust B**. This trust can be misplaced.

Classic example: An email client delegates to a full-featured editor (vi, emacs) for composing messages, but editor provides shell access.

Better: Use a restricted editor, or sandbox the editor so shell escape runs with reduced privileges.

Transitive Trust in Our Examples

System	Third-party Component	What You're Trusting It With	If Compromised
CMS	Plagiarism detection	Every student submission	Document exfiltration
CMS	LTI plugin	User session, course data	Account takeover
Payment app	Push notification SDK	In-memory auth tokens	Token theft
Payment app	Analytics library	Transaction metadata	Privacy breach

For every third-party component in your DFD:

What am I trusting this with? What's the blast radius if it's compromised?

Compartmentalization and Sandboxing

Isolate components in compartments so that compromise of one doesn't give access to others.

Mechanism	Granularity	Example
Process isolation	Per-component	Chrome: each tab in a separate process
seccomp-bpf (Linux)	Per-syscall	Restrict process to read, write, exit only
Containers / VMs	Per-service	Microservice isolation
Mobile sandboxing	Per-app	iOS/Android sandbox each app by default

Promote Privacy

Restrict the flow of sensitive data as much as possible, reducing the trust surface.

Data minimization:

- Don’t collect data you don’t need
- Don’t store it longer than you need
- Don’t transmit it to components that don’t need it

System	What the component receives	What it should receive
CMS → Plagiarism service	Student name, email, ID, submission	Text content with anon ID only
Payment app → Analytics	Full transaction details	Aggregated statistics only

Restricting Data Views: Least Privilege Again

Example: A student admission system receives sensitive letters of recommendation as PDFs.

- **Typical design:** Reviewers download PDFs to local machines → compromise of those machines leaks private information
- **Better:** PDFs viewable only in a secure browser viewer; no data downloaded to the client

Payment app: Customer support agents investigating a dispute:

- Default: show masked data (`****1234`)
- Full access: requires supervisor override + audit log entry

Input Validation as Trust Minimization

Input validation is a form of trust with reluctance: trust a subsystem *only under specific circumstances*.

- Trust a function's parameters only if they're within expected range (e.g., buffer length)
- Trust a web form field only if it contains no code-interpretable strings (XSS, injection)
- Trust a serialized object only if it contains no executable content (deserialization attacks)

Apply at every trust boundary in the DFD.

Monitoring and Traceability

Designing for Detection

If you are attacked, **how will you know?**

Once you learn, **how will you discern the cause?**

Software must be designed from the start to produce operational telemetry.

Detecting Attacks in Progress

These patterns are **not detectable from a single log line**. They require *aggregation* and *correlation*.

Payment app anomalies:

- Many small transfers to new accounts
- Transfers at unusual hours from unusual locations
- Sudden spikes in failed login attempts
- Transfers just under reporting thresholds ($9,999 instead of $10,000)

CMS anomalies:

- A single account downloading every student's submission
- Repeated authorization failures (probing for IDOR)
- Grade changes outside normal grading periods

Log Aggregation and Correlation

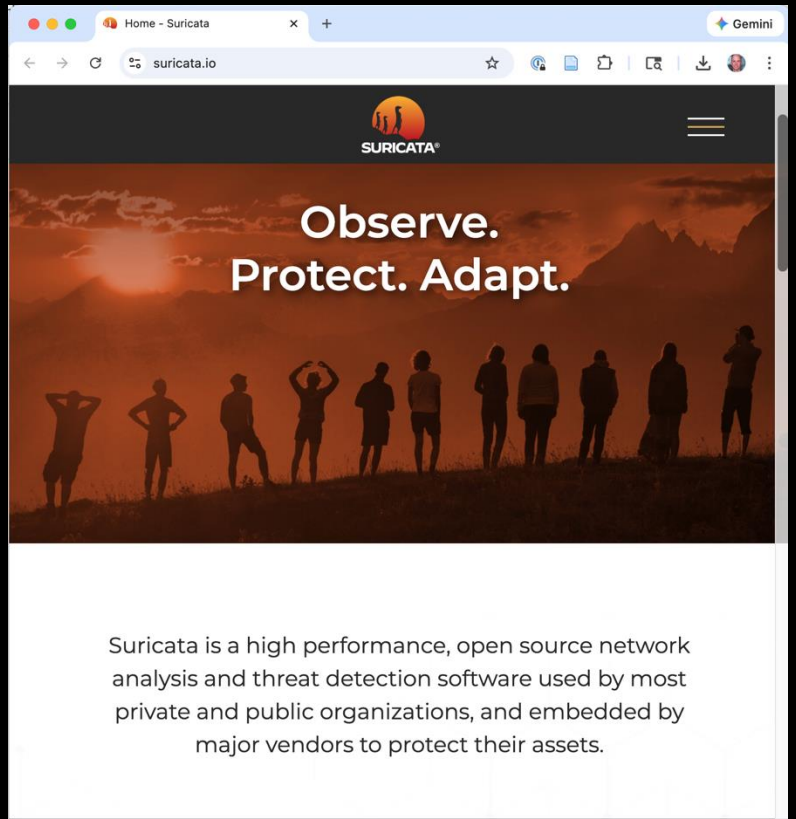
In a distributed system, evidence is spread across multiple services. A single attack may touch auth, payment, notifications, etc.

Correlation example:

User X's session created at 3:42am from an IP in a new country
→ three transfers totaling $9,999
→ push notifications silenced

Design implication: Include **correlation IDs** in every log entry.

Intrusion Detection and Log Analysis



Pulling It Together

From Threat List to Architecture

Individual threats often converge on the **same mechanism**.

A security architecture is *not* “for threat X, do Y” repeated 20 times. It’s a coherent design where mechanisms serve multiple purposes.

CMS convergence:

- Spoofing of student identity + elevation of privilege to instructor + information disclosure of grades → **centralized, relationship-based access control with server-side enforcement and complete mediation**

Payment app convergence:

- Tampering with transfer amount + spoofing of sender + disclosure of bank credentials → **authenticated encryption in transit, per-transaction re-authentication, encrypted credential storage with KMS**

The Principles as a Design Review Checklist

Principle	Review Question
Favor simplicity	Is the security-critical code path as simple as possible?
Fail-safe defaults	Does the system deny by default? Are defaults secure?
Don’t expect expert users	Does security depend on unreliable user behavior?
Small TCB	What *must* be correct, for security? Is that set minimal?
Least privilege	Does every component have minimum access needed?
Compartmentalization	If component X is compromised, what else is exposed?
Promote privacy	Are we collecting/storing/sending more data than needed?

Design Review Checklist (continued)

Principle	Review Question
Complete mediation	Is every access checked, every time, at the server?
Separation of privilege	Does any critical action depend on a single condition?
Open design	Does security depend on obscurity, or on keys and controls?
Defense in depth	If one layer fails, does another independent layer catch it?
Monitor and trace	If attacked, will we know? Can we reconstruct what happened?

Three Categories as a Completeness Check

Category	Have we addressed it?
Prevention 	Eliminating classes of flaws (type-safe languages, vetted crypto, input validation, simple design)
Harm Reduction 	Containing damage from unknown flaws (least privilege, compartmentalization, privacy, separation of privilege)
Detection & Recovery 	Knowing when something goes wrong (audit logging, monitoring, correlation, alerting)

If any category is empty, the architecture has a gap.

Part 2 Summary

For each mitigation, ask: does this depend on individual developer vigilance, or is it enforced by the ecosystem?

Topic	Key Techniques	Principles
Authorization	RBAC, ABAC/ReBAC, Cedar, server-side enforcement	Complete mediation, Least privilege
Integrity	Input validation (allowlists), HMAC, authenticated encryption	Fail-safe defaults, Trust w/ reluctance
Accountability	Audit logging, correlation IDs	Monitor and trace
Trust minimization	Small TCB, transitive trust analysis, code signing	Trust w/ reluctance, Simplicity
Compartment-alization	Process isolation, seccomp-bpf, containers	Separation of privilege
Privacy	Data minimization, restricted views	Promote privacy
Architecture	Convergence of mitigations	Defense in depth

Quick-Reference: Secure Design Techniques

Technique	Principle(s)	CMS	Payment App
TLS / HTTPS	Open design, DiD	Grades in transit	Transfer instructions
AES-GCM at rest	DiD, Privacy	Student records (FERPA)	Bank account numbers
Argon2id	DiD, Fail-safe	Account passwords	Account passwords
IdP / SSO	Trust w/ reluctance	University SSO (SAML)	Auth0/Okta for MFA
mTLS	Trust w/ reluctance	LTI plugin integration	Server ↔ bank API
Cedar policies	Simplicity, Small TCB	TA scoped to section	User-owns-account
Compartmentalization	Compartmentalize	PDF renderer sandboxed	Payment ≠ social feed
Audit logging	Monitor and trace	Grade change history	Transfer evidence

References

- Kern, C. “Developer Ecosystems for Software Safety.” *Communications of the ACM* 67, no. 6 (June 2024): 52–60.
- Saltzer, J.H. & Schroeder, M.D. “The Protection of Information in Computer Systems” (1975)
- CIS/SAFECode, “Secure by Design: Guide to Assessing Software Security Practices” (2025)
- Cedar Policy Language – https://www.cedarpolicy.com/
- Schneier, B. “Attack Trees” (Dr. Dobb’s Journal, 1999)
- OWASP Authorization Cheat Sheet
- OWASP Input Validation Cheat Sheet
- Anderson, J.P. “Computer Security Technology Planning Study” (1972) – the reference monitor concept
- NIST SP 800-218: Secure Software Development Framework (SSDF)