EDUSERV IT · IAM PLAYBOOK

AUTHZ UNLOCKED

Authorization Playbook

Authorization in IAM — 8 Sections for Beginners & Professionals

WHAT'S INSIDE — AT A GLANCE

This cheat sheet covers 8 essential sections on Authorization in IAM — from first principles to real systems, common mistakes, and interview readiness.

01	**Definition**	What authorization is — and why it matters
02	**Why It Exists**	The real access control problem it solves
03	**Real-World Flow**	Step-by-step: how it works in a live system
04	**Key Components**	Every building block involved in authorization
05	**Real-World Scenario**	An HR portal login, explained behind the scenes
06	**Common Mistakes**	High-value lessons engineers get wrong
07	**Architecture View**	Where authorization sits in your system
08	**Interview Power Points**	Key facts expected in IAM interviews

- **AuthZ = deciding WHAT you can do**
- **RBAC = access by role**
- **Least Privilege = minimum access needed**
- **Policy = the rule that decides yes or no**

01 DEFINITION

"What is Authorization?" — The gatekeeper after identity is confirmed

DEFINITION

Authorization determines what an authenticated identity is allowed to do — read, write, delete, or admin.

KEY CONCEPTS

- **Authorization** — Decides WHAT an authenticated user can do — not who they are.
- **AuthN vs AuthZ** — AuthN = proving identity (WHO). AuthZ = controlling access (WHAT). Always in that order — two completely separate steps.
- **Permission** — A specific allowed action on a resource: "read:invoices", "delete:user", "write:config".
- **Role** — A named collection of permissions assigned to a user: Admin, Developer, Viewer.
- **Policy** — A rule that defines who can do what under which conditions — the decision logic.
- **ACL (Access Control List)** — A list specifying which users or roles can access a specific resource. More granular than roles alone.

REAL-WORLD EXAMPLE

— **Priya logs into the company HR portal.**

— **Authentication confirmed WHO she is.**

— **Authorization now decides WHAT she can do** — view payroll? Edit records? Access the admin panel?

— **These decisions are controlled entirely by authorization policies** — assigned to her role.

WHY IT MATTERS IN INDUSTRY

— **Authorization is the #1 control** — preventing privilege abuse inside organisations.

— **Misconfigured permissions** — are a leading cause of insider threats and data breaches.

— **Cloud platforms (AWS, Azure, GCP)** — are entirely permission-driven — authz is everything.

— **Zero Trust architecture** — demands explicit authorization on every single request.

02 WHY AUTHORIZATION EXISTS

The real access control problem it solves

DEFINITION

Without authorization, every authenticated user could access everything — data, admin controls, and all systems.

KEY CONCEPTS

- **The Problem** — Authentication only confirms identity. It gives no information about what the user should be allowed to access.
- **The Risk** — Without authz, a junior employee could read CEO emails, delete databases, or access HR records.
- **The Solution** — Authorization enforces boundaries: each identity gets only the access it legitimately needs.
- **Scope** — Applies to every resource: APIs, databases, files, dashboards, admin panels, cloud services.
- **Foundation** — Every compliance framework (SOC2, HIPAA, ISO 27001) requires formal access controls.

REAL-WORLD EXAMPLE

— **A bank authenticates every employee at the entrance.**

— **Without authorization, a teller could access** — loan approvals, wire transfer controls, or executive accounts.

— **Authorization ensures the teller can only access** — what their role requires — customer views, nothing more.

WHY IT MATTERS IN INDUSTRY

— **Privilege abuse by insiders** — is a top cause of data breaches — authorization limits the blast radius.

— **Regulatory compliance (HIPAA, GDPR, PCI-DSS)** — requires demonstrable access control on sensitive data.

— **Cloud environments have thousands of resources** — without authz, one compromised account = total access.

— **The Principle of Least Privilege** — is a core security mandate in every major security framework.

03 REAL-WORLD AUTHORIZATION FLOW

Step-by-step — How it actually happens in a live system

DEFINITION

Authorization follows a structured flow: authenticated user makes a request → PDP evaluates roles and policies → PEP enforces the decision.

KEY CONCEPTS

- **Step 1 — User Request** — Authenticated user clicks a resource, calls an API, or opens a dashboard.
- **Step 2 — Token Verified** — System verifies the JWT token — confirms WHO the user is before checking WHAT they can do.
- **Step 3 — PDP Evaluates** — The PDP retrieves the user's role and permissions, then evaluates policies: Does this role have permission to perform this action on this resource? Returns allow or deny.
- **Step 4 — PEP Enforces** — PEP enforces the PDP decision at the API gateway. ALLOW → resource returned. DENY → 403 Forbidden returned.

REAL-WORLD FLOW

1	**User Makes a Request** *User*	Authenticated user clicks a resource, calls an API, or opens a dashboard.
2	**Token Verified** *IdP / Token*	System verifies the JWT token. Confirms WHO the user is before checking WHAT they can do.
3	**PDP Evaluates Request** *PDP — Policy Decision Point*	The PDP retrieves the user's role, evaluates policies, and returns allow or deny. Role lookup and policy evaluation both happen inside the PDP — one component, not two.
4	**PEP Enforces Decision** *PEP — Policy Enforcement Point*	PEP enforces the PDP decision at the API. ALLOW → resource returned. DENY → 403 Forbidden returned.

KEY INSIGHT	The PDP is the brain — retrieves roles, evaluates policies, returns allow/deny. The PEP is the gatekeeper — enforces the PDP decision at every API call. PEP never decides, PDP never enforces. Authorization ALWAYS happens after authentication — never before.

04 KEY COMPONENTS INVOLVED

The building blocks of every authorization system

DEFINITION

Authorization involves multiple components working together — not just a role check on a button.

KEY CONCEPTS

- **User / Principal** — The authenticated identity requesting access — human, service, or machine.
- **Application** — The system that receives the request and triggers the authorization check.
- **Role** — A named group of permissions assigned to users: Admin, Editor, Read-Only.
- **Permission** — A specific action on a specific resource: "read:payroll", "write:config".
- **Policy** — Maps roles + attributes to allowed/denied actions. Explicit DENY always overrides ALLOW — even if another policy grants access.
- **PDP — Policy Decision Point** — Retrieves roles, evaluates policies, returns allow/deny. Role lookup and evaluation both happen inside the PDP — one component, not two.
- **PEP — Policy Enforcement Point** — Gatekeeper at every API. Receives the PDP decision and enforces it. Never makes its own decisions.
- **Resource / Protected Data** — The asset being accessed: API endpoint, database record, file, or dashboard.

REAL-WORLD EXAMPLE

— **A Developer calls the GitHub API to push code.**

— **GitHub passes the JWT token to the PDP.**

— **PDP checks: does "Developer" have "write:repo" permission?** — Policy says YES — PEP allows the push.

— **A Viewer role returns DENY.** — PEP returns 403 Forbidden.

— **An explicit DENY on a protected branch** — overrides any allowing policy — deny always wins.

WHY IT MATTERS IN INDUSTRY

— **Separating roles from permissions** — allows fine-grained control without rewriting code.

— **Explicit DENY overrides ALLOW** — critical for AWS IAM, Azure RBAC, and OPA.

— **PDP/PEP separation is the industry standard** — XACML, OPA, and AWS IAM all use this model.

— **Every cloud resource (S3 bucket, Lambda, RDS)** — has its own permission model — know them all.

05 REAL-WORLD SCENARIO

Employee accessing an HR Portal — What happens behind the scenes

SCENARIO

1	**User logs in successfully** *Authentication / IdP*	Priya logs in. IdP verifies credentials and issues a JWT token. Her role may be in the token OR looked up dynamically by the PDP — both patterns exist in real systems.
2	**Priya opens the HR Portal** *Application*	She clicks "View Payroll". The application receives her request with the JWT token attached.
3	**PDP evaluates the request** *PDP — Policy Decision Point*	PDP retrieves her role (HR-Staff) and evaluates: "HR-Staff CAN read payroll. HR-Staff CANNOT edit payroll." Request = read. Decision = ALLOW.
4	**PEP enforces the decision** *PEP — Policy Enforcement Point*	The PEP gates the API. Since PDP returned ALLOW, the payroll API is called and data is returned to Priya.
5	**Priya tries to edit payroll** *Privilege Check*	She clicks "Edit Salary". PDP re-evaluates: HR-Staff cannot edit payroll. PEP blocks the request. 403 Forbidden returned.
6	**Manager logs in separately** *RBAC in Action*	"HR-Manager" role has edit permissions. The same policy engine returns ALLOW for the manager's edit request. Same system, different role, different outcome.
7	**Explicit deny scenario** *Deny Override*	Even if a broad policy granted HR-Staff edit access, an explicit DENY on payroll editing overrides it. Deny always wins — no exceptions.
8	**All decisions are logged** *Audit / SIEM*	Every allow/deny decision is written to the audit log: who, what, when, and the decision. Compliance requirement — not optional.

06 COMMON MISTAKES

High-value lessons — what beginners and engineers get wrong

×	**Confusing AuthN with AuthZ**	AuthN = proving WHO you are. AuthZ = deciding WHAT you can do. Always two completely separate steps — never combine them in your mental model or your code.
×	**Giving excessive permissions**	Over-provisioning is the #1 authz failure. Always grant the minimum permissions required — nothing more. Every extra permission is an attack surface.
×	**Ignoring Least Privilege**	"Give access just in case" is dangerous thinking. Least privilege is not optional — it is a core security mandate in every compliance framework.
×	**Permanent admin access**	No one should have permanent admin. Use Just-In-Time (JIT) access: elevate temporarily when needed, revoke automatically after use.
×	**Skipping API-level checks**	Hiding a button in the UI is NOT authorization. The API endpoint must validate permissions independently on every single call — UI state is never a security control.
×	**Assuming roles always live in the token**	Roles can be in the JWT token OR looked up externally at decision time. Assuming roles are always in the token causes bugs when roles change mid-session.
×	**No access reviews**	Permissions accumulate silently as people change roles. Regular access reviews catch stale permissions before attackers exploit them.
×	**Ignoring fail-closed principle**	When the PDP is unavailable, deny all requests by default — fail closed. Failing open when auth is down is a critical security vulnerability.

07 ARCHITECTURE VIEW

Where authorization sits in your system

◆ **After Authentication**	AuthZ sits immediately AFTER auth. No valid token = no authz check. Always in that order — never skip.
◆ **Before Resource Access**	Every API endpoint, database query, and UI action is gated by the authz layer before any data is returned.
◆ **PEP — Enforcement**	Policy Enforcement Point: gatekeeper at every API. Blocks or allows based on the PDP decision. Never makes its own decisions.
◆ **PDP — Decision**	Policy Decision Point: the brain. Retrieves roles, evaluates policies, returns allow/deny. If PDP unreachable — fail closed, deny by default.
◆ **PAP — Admin**	Policy Administration Point: where policies are written, stored, and managed. A change here instantly affects all users.
◆ **Coarse vs Fine-Grained**	Coarse-grained authz controls broad access — can this user reach this API at all? Fine-grained controls specific records — can this user read this specific employee's salary? Both layers are needed in production systems.
◆ **Audit Log**	Every authz decision — allow and deny — is logged: who, what, decision, and when. Essential for compliance and forensic investigation.

08 INTERVIEW POWER POINTS

Key facts — expect these in IAM interviews

Q1	**Authorization (AuthZ) = deciding WHAT an authenticated identity is allowed to do. Always happens after AuthN — never before.**
Q2	**RBAC (Role-Based) = access by role (Admin, Developer, Viewer). Most common model in enterprise systems.**
Q3	**ABAC (Attribute-Based) = access by attributes: IF department=Finance AND location=HQ AND time=business_hours THEN allow:read:salary. More flexible than RBAC for complex conditions.**
Q4	**Least Privilege = grant only the minimum permissions needed. Every extra permission is an attack surface — non-negotiable.**
Q5	**PDP = the brain that evaluates policies and returns allow/deny. PEP = the gatekeeper that enforces it. PEP never decides, PDP never enforces — always separate.**
Q6	**Explicit DENY always overrides ALLOW. If two policies conflict, the deny wins. Critical for AWS IAM, Azure RBAC, and OPA.**
Q7	**JIT Access (Just-In-Time) = temporary elevated permissions granted on demand, automatically revoked after use.**
Q8	**Fail closed = deny all requests when PDP is unavailable. Always design for fail closed — never fail open.**

MEMORY HOOK

Authorization = Deciding which rooms you can enter after passing the security gate. You've proven WHO you are (AuthN) → Now the system decides WHICH doors you can open, WHAT you can touch, and WHAT is locked — that's AuthZ. The PDP decides. The PEP enforces. Deny always wins.