RFC

Fine-Grained Authorization in the Grafbase Enterprise Platform

Contents

Tom Houlé (Grafbase) — 2025-09-23

1. Introduction

There is demand for finer grained authorization in the Grafbase Enterprise Platform. This RFC is an attempt at a principled, future-proof solution that will give us configurable, robust and performant access control that is maximally convenient for our users and does not require any additional infrastructure or setup as part of the Enterprise Platform.

The scope of this RFC is authorization in the Enterprise Platform:

- for human users logging in with the Dashboard or CLI,
- for non-human agents interacting with the Grafbase API, object storage and the telemetry sink. For example CI jobs, Grafbase Gateway, external integrations or AI agents.

This RFC will often reference OpenID Connect (OIDC) and OAuth 2.0 concepts, basic familiarity with them is assumed. Our current authentication solution is based on OIDC.

2. Requirements

2.1. Infrastructure simplicity

We want to avoid introducing additional infrastructure or setup as part of the Enterprise Platform. Fewer, simpler services mean a faster path to adoption, easier security reviews and generally a platform that is easier to operate.

2.2. Performance

Access control should not introduce excessive latency or network roundtrips.

2.3. Familiarity and convenience

As much as possible, the authorization model should be familiar to enterprises looking to adopt Grafbase. It should be documentable, and the path to customization should be clear and accessible.

2.4. Expressive policies

We want to enable fine-grained authorization decisions based on ownership of resources, team memberships, roles and other attributes. If the design does not accommodate this requirement, we will need to tack on more configurability to the solution later as a workaround — we should have enterprise use cases in mind from day one.

2.5. Customizability

Different organizations want different access control rules. We should have defaults, but everything must be tweakable to the needs of enterprises adopting Grafbase.

2.6. Suitability to both the hosted and self hosted Enterprise Platform

Self-hosting is top of mind, but the new solution should also work for the multi-tenant deployment of the Grafbase Enterprise Platform at grafbase.com.

2.7. Integration

Enterprises have centralized authentication, internal user management and authorization services — we will dive into that in more detail in this document. We should support delegating to these services.

3. Landscape

3.1. Terminology

- In an access control decision, a **principal** is the subject of the decision. The principal is who or what wants to perform an action on a resource. Sometimes, a distinction is made between principal and **subject**, where the subject is a representation of the principal. For example, if the principal is a user, the subject could be that user's user ID in the system.

  > A security principal is any entity that can be authenticated by the operating system, such as a user account, a computer account, or the security groups for these accounts.
  >
  > — Microsoft Learn

- The **action** is the operation that the principal wants to perform on the resource. There are standard actions such as "read", "write", and "delete", but many relevant access control actions do not fit in these standard verbs.
- A **resource** is either a category of objects ("organization settings", "graphs") or an object ("user", "team", "graph") that is the object of an access control decision.
- The **context** is the additional information that is required to make an access control decision. This can include the time of day, the location of the user, or the device being used.

3.2. Models of authorization

- **Role-based access control (RBAC)** is the traditional model for authorization. Access is determined by roles. You assign permissions to a role (like editor or viewer), and then you assign users to those roles. A user gets all the permissions associated with their assigned role(s). In pure RBAC, categories of users (roles) have access to specific actions on categories of resources: "admins can edit teams", "users with the engineer role can publish subgraphs". There is no notion of ownership ("I am allowed to see my patients' medical records"), conditional access ("employees can edit this data only during business hours") or sharing ("I can view the documents shared with me by their owners") in pure RBAC.
  - RBAC usually supports a hierarchical structure, where roles can inherit permissions from other roles. This allows for more flexibility and scalability.
  - RBAC is susceptible to role explosion, as soon as you have too many combinations of permissions to express in a small number of roles.
  - In real world scenarios, RBAC is often extended with some notion of ownership of resources, but often in ad-hoc, application specific ways.
- **Attribute-based access control (ABAC)** is more dynamic and granular. Access decisions are made by evaluating policies based on attributes (characteristics) of the principal, the resource they are trying to access, and the environment (like time of day or location). For example, a policy might state: "Allow a manager to access only their reports' performance reviews, and only during business hours.". Attributes can include ownership information, such as the user's department or the resource's creator.

    - In ABAC, roles can be expressed as an attributes of the principal.
- **Relationship-based access control (ReBAC)** grants access based on a principal's relationship to a resource. It answers the question, "Can this user perform this action on this resource because of how they are connected?" For example, you can edit a Google Doc because you are its owner, or you can see a photo on Instagram because you are a follower of the person who posted it. Relationships are rule based and transitive. This is a very powerful model for applications like Google Drive, where ownership can be shared ("I allowed this external user to view, but not edit, that document"), inherited ("ownership of a directory grants you ownership of its subdirectories"), or at the intersection of multiple rules, for example when a user belongs to a company-wide group that has `Viewer` access to a folder, but they have also been given a direct link that grants `Commenter` access to a specific document inside it. The system must resolve both the inherited 'viewer' rule from the folder and the direct 'commenter' rule from the link to determine the user's final permission on that document. ReBAC is also known as the Zanzibar model, from the Google paper that introduced it.

Modern enterprise authorization implementations use elements of all these models, where roles are still a relevant concept, layered in with attribute-based access control and fine-grained permissions for each instance of a resource. Even ostensibly ReBAC focused systems like SpiceDB and OpenFGA bake in a context of attributes (ABAC) when evaluating access decisions. These models complement each other. They can coexist in the same system.

Another term that has become common is **Policy-based access control (PBAC)**. In PBAC, a policy decision point evaluates policies to manage what an individual or system can do. These policies are essentially a set of rules and conditions that are centrally managed and can be as simple or as complex as needed. Instead of permissions being hard-coded into applications, PBAC externalizes these decisions, allowing for greater flexibility and easier management of access rights. That most often goes hand in hand with policy as code (see below). While RBAC, ABAC, and ReBAC represent different methodologies for determining access, PBAC can be seen as an overarching approach that often incorporates elements of these other models to enforce its policies.

3.3. Modern enterprise authorization

3.3.1. Policy as code

Authorization as Code (AaC) is the practice of managing and defining access control policies using code, rather than through manual configurations in a database or an admin UI. These policy files are treated just like application code: they are stored in version control (like Git), tested automatically, and deployed through a CI/CD pipeline. It is a subset of Policy as Code.

The auditability, traceability and composability benefits known from the usage of code and VCSs as sources of truth apply here in the same way as in other models like Infrastructure as Code (IaC).

In this model, authorization logic is also centralized and decoupled from application code: rules are managed in one central place instead of being scattered across one or more application's codebase.

Policies can be automatically tested for errors and vulnerabilities before they go live, dramatically reducing the risk of misconfigurations that could lead to data breaches. This makes the system easier to understand, manage, and update. Collaboration is also simplified:writing policies as readable code allows different stakeholders—from developers to security and product managers—to understand and collaborate on the rules that govern their application.

Examples:

- IAM configuration with Cloudformation or Terraform
- Rego policies in Open Policy Agent (OPA)
- SpiceDB schemas

3.3.2. Integration with OAuth 2.0 and OpenID Connect

Open, well-supported standards that enable interoperation, in particular OAuth 2.0 and OpenID Connect (OIDC), are preferred to application-specific authentication and authorization mechanisms.

The standard set up in enterprises is a federated authentication with a central identity provider (IdP).

3.3.3. Stateless authorization

Where possible, stateless authorization models, where all the necessary context and permissions are embedded in JWT claims, is preferred. Enterprise IdPs like Okta allow organizations to provision and manage custom scopes and fine-grained authorization data with Rich Authorization Requests (RAR) directly inside users' JWTs for a given application. That configuration stays inside the IdP instead of inside separate applications with their separate state.

3.3.4. Emerging standard: AuthZEN

AuthZEN is a standardization effort started in 2023 and hosted by the OpenID foundation. AuthZEN wants to be to authorization what OpenID Connect is to user authentication.

The actors and their roles are described by the **P*P model**:

- **Policy Enforcement Point (PEP)**. The protected resource server. That is to say, the service that sits in the critical path of application requests. In our case, the Grafbase API. This role is often played by API Gateways. That issue is orthogonal, but we are exploring fleshing out support for acting as an AuthZEN PEP in Grafbase Gateway.
- **Policy Decision Point (PDP)**. The authorization service. When a PEP receives a request, it sends a query to the PDP. The PDP then makes the access decision by evaluating the request against the policies it receives from the PAP and the data it gathers from the PIP. The PDP can also act as both PAP and PIP, depending on the implementation.
- **Policy Information Point (PIP)**. The PIP provides the PDP with the necessary context and attributes to make a decision. This can include information about the user (like

their session validity), the resource, the environment, and other business-related attributes, in addition to the context provided by the PEP.
- **Policy Administration Point (PAP)**. The PAP is responsible for creating, managing, and updating the access control policies that the PDP will use. This allows for centralized control over authorization rules.

In its current state (Implementer's Draft 1), the spec is only about the interaction between PEP and PDP. It defines a standard interface between the two.

> The Authorization API described in this document enables different providers to offer PDP and PEP capabilities without having to bind themselves to one particular implementation of a PDP or PEP
>
> — AuthZEN spec

It defines one endpoint on the PDP that returns access control decisions based on a subject (representation of the principal), action, resource and context. The draft spec itself is short, easy to read and implement.

In this RFC, we will also focus on just the PEP and PDP. The PEP will be the Grafbase API. The PDP will be either embedded in the Grafbase API, or an external service, depending on the configuration of a specific Enterprise Platform deployment.

While the PDP-PEP separation is seeing increasing adoption in enterprise environments, it is uncommon for access control for entire third-party applications to be delegated to an internal PDP, because of the large amount of special policies to write, and the need to reflect an internal data model that, by definition, is developed by third-parties.

3.3.5. Workload identity

Workload identity is the concept of giving a unique, verifiable identity to an automated process, like a CI/CD pipeline or a microservice, allowing it to authenticate and access resources securely without relying on long-lived, static secrets like API keys. This approach significantly enhances security by moving away from secrets that can be leaked or compromised. Instead of storing a secret, the workflow can prove its identity to another system, such as a cloud provider, or a service like the Grafbase API, and receive a short-lived access token in return. This is often accomplished using OpenID Connect (OIDC). In this M2M (machine-to-machine) context, the CI/CD platform acts as an OIDC provider, issuing an identity token that describes the workflow's context (e.g., repository, branch, commit SHA). The resource provider can then be configured to trust this OIDC provider and exchange that identity token for a temporary cloud access token. This is precisely how platforms like GitHub Actions and CircleCI enable passwordless authentication to services like AWS, Azure, and Google Cloud, making pipelines more secure and manageable.

These concepts are being generalized in frameworks like SPIFFE (Security and Privacy Identity Framework for Everyone).

See this good Microsoft docs page on an example of how to use workload identity with Azure.

4. Proposed solution

Our users' use cases exist on a spectrum:

- On one end, users of the managed platform and organizations in the earlier stages of adoption will want a built-in authorization model, without deep integration with their IdP, either because their IdP does not give them the power to grant fine grained entitlements and memberships, or because they prefer having everything in the same UI (our dashboard).
- On the other end, enterprises who want to rely on their central, powerful identity provider (IdP) for fine-grained entitlements, custom scopes, centralized team memberships. These organizations may also already have implemented an external service for authorization, like Open Policy Agent (OPA).

We should accomodate all users between these two extremes.

Furthermore, as hinted above, individual organizations' use cases will differ in two other ways:

1. Organizations will want enforcement of access control to be baked in the Grafbase Enterprise Platform, others will want to rely on their central policy decision point (PDP).
2. Organizations will want to lean more or less on their IdP for membership and ownership relations. Some will want to base authorization decisions on fine-grained permissions embedded in ID tokens by their IdP, with custom scopes and Rich Authorization Requests (RAR). Others will want to configure individual users and teams' permissions in the Grafbase Dashboard. This is a spectrum, with one extreme not making use of the data in the Grafbase Database for authorization purposes, and the other, more common extreme having no relevant data except user identity in their users' JWTs.
   - This point is only about human users. For machine-to-machine use cases, we will have a single way to embed fine-grained entitlements in the access tokens (see below).

For (1.), the solution is an interface used by the Grafbase Enterprise Platform to request authorization decisions, so the PDP is pluggable, with a default implementation baked into the `api` image of the Enterprise Platform. This RFC proposes using the AuthZEN Access Evaluation API as that interface.

To address (2.), the application will send **both** the relevant claims from the JWT and the relevant data from the Grafbase API Database. The PDP on the other side of the interface is responsible for taking the context it deems relevant into account.

To reconcile these differing needs, the rest of the RFC will expose a solution that can be summarized as:

- on the **entitlements/context/claims** side, *both* built-in roles, teams and ownership data, and custom scopes / RARs from the IdP
- on the **access control** side, a built-in Cedar Policy based decision point (PDP) with the ability to replace with or augment with an arbitrary PDP, with communication to that arbitrary PDP taking place as AuthZEN HTTP requests. What authorizer implementation is used will be a matter of configuration.

We will touch first on the interface between the application layer and the PDP, then on the built-in PDP using Cedar Policy, and finally, on a new access tokens implementation with fine-grained permissions baked in.

4.1. The interface

4.1.1. API

The interface between the application and authorization layers is modeled after the AuthZEN Access Evaluation API. Just like in AuthZEN, there will be a single decision method and a batch method. In the Grafbase API, it will be modelled as a Rust trait. In pseudocode:

```
type Properties = serde_json::Map;

struct Subject {
    id: String
    r#type: String,
    properties: Option<Properties>,
}

struct Action {
    name: String,
    properties: Option<Properties>,
}

struct Resource {
    id: String,
    r#type: String,
    properties: Option<Properties>,
}

struct Request {
    subject: Subject,
    action: Action,
    resource: Resource,
    context: Option<serde_json::Map>,
}

struct Decision {
    decision: bool,
    reason: Option<String>,
}

pub(crate) trait Authorizer {
    async fn authorize(&self, request: &Request) -> Result<Decision, Error>;
    async fn batch_authorize(&self, requests: &[Request]) -> Result<Vec<Decision>,
Error>;
}
```

We will want more precise types in `Request`, to keep track of what data we actually send and keep it consistent.

We will have two implementers of these traits:

- `HttpAuthzenAuthorizer`, which forwards the requests to an AuthZEN compliant server over HTTP, and returns the decision(s).

- `CedarAuthorizer`, which is described below.
- `HttpAuthzenAndCedarAuthorizer`, which will combine the decisions from both `HttpAuthzenAuthorizer` and `CedarAuthorizer`.

Which implementer is used is based on configuration, either in a structured way if we introduce a configuration file, or through environment variables, for example `GRAFBASE_AUTHZEN_HTTP_PDP_URL`.

4.1.2. Schema and entities

The trait above defines the shape in which authorization requests and the decisions in their responses are communicated. For external PDPs to effectively serve as authorizers for the Grafbase Enterprise Platform, they must know what to expect. We must define and document:

- The entities that can appear as subjects and resources.
- The actions and their properties.
- The shape of the context.

Except for the shape of the context, the Cedar schema format is a very readable language for describing this kind of schemas. It has a JSON mapping which would let us generate code, docs and forms in the dashboard directly from the schema, keeping a single source of truth for our authorization logic's data types and actions. We will elaborate more on Cedar below.

You can find a work-in-progress schema for the Grafbase Enterprise Platform authorization logic at grafbase/cedar-policies on GitHub.

4.2. Built-ins

4.2.1. Built-in data model

At minimum, a PDP must know which resources it controls access to. In our domain, these would be:

- Organizations
- Users
- Teams
- Graphs
- Branches
- Subgraphs
- Schema Proposals

All these resources live in the Grafbase API's database. Organizations, users and teams can be just-in-time provisioned to match groups and users in an IdP. We already have that logic for organizations and users, we would only need to add it for teams.

We can keep the existing data model for teams, with a tree hierarchy of nested teams.

Graphs, branches and subgraphs can be owned by teams.

Schema proposals can have multiple authors (users or teams), and multiple reviewers.

All the ownership data mentioned in this section must be available as properties on the principal, resources or context when requesting an authorization decision.

Different actions on each of these resources are modeled as different actions in the authorization schema.

4.2.2. Roles

In the Grafbase API database schema, organizations have members, which currently can have one of three roles: “owner”, “admin” or “member”. The only difference between an admin and an owner is that the owner can delete the organization. For maximum flexibility, this RFC proposes:

- Removing the “owner” and “member” roles. By default, there are only admins and regular members. Admins have permissions to modify the organization itself and manage teams, while members can work within the organization, create graphs, etc. These roles, combined with ownership of graphs, branches and subgraphs by teams, offers a good balance of flexibility and restrictions on individual members.
- Allowing admins to define organization-specific roles and assign them to users or teams. These roles will be passed as part of the principal’s properties in authorization requests, so they can be taken into account in access control decisions.

A role is a collection of permissions. Permissions are coarse-grained action-resource pairs. An example role, represented as JSON, may look like this:

```json
{
  "name": "SubgraphPublisher",
  "permissions": [
    "can_publish:subgraph",
    "can_create:subgraph",
    "can_edit_schema_check_settings:branch"
  ],
}
```

The purpose of roles is to provide a first level of coarse grained access control. Fine grained access control comes when roles are combined with ownership over resources.

4.2.3. Built-in PDP with Cedar

For easy integration adoption, we need a good out-of-the-box solution for authorization. That solution has to play well with the subject-action-resource-context model, and it has to be embeddable in the Grafbase API deployment.

Zanzibar-like (ReBAC) solutions like SpiceDB and OpenFGA are great, but they are not a fit here because they require much deeper integration in the application (syncing relations to a database).

PDPs that are deployable as sidecars would be viable: Cerbos and Open Policy Agent (OPA) are the notable options here.

Writing our own policy engine is also an option, but it is more work, and it is not our core business domain.

There is a solution that is well supported, embeddable in Rust, declarative, fast, well documented, and easy to use: Cedar Policy, a policy and schema language developed by AWS for its Amazon Verified Permissions product. The whole language and engine are open source — licensed under Apache 2.0 — and available on crates.io in the cedar-policy crate.

Cedar’s authorization model is simple: every action is forbidden by default. You write policies that allow some actions for some principals on some resources, with context. These policies can look like this:

```
permit (
    principal in Group::"janeFriends",
    action in [Action::"view", Action::"comment"],
    resource in Album::"janeTrips"
);
```

Or `forbid` policies, that override `permit` rules wherever the `forbid` rules apply. You can add conditional logic with `when` and `unless`:

```
// Only the owner can access any resource tagged "private"

forbid ( principal, action, resource )
when { resource.tags.contains("private") }    // assumes that resource has "tags"
unless { resource in principal.account };     // assumes that principal has "account"
```

If a type error occurs anywhere when evaluating a policy, that policy does not apply.

The schema language is also very straightforward. It has only entities, actions, type aliases and namespaces:

```
entity Album {
  owner: User,
  flags: {
    organizations?: Set<Org>,
    locales?: Set<Location>,
    tags: Set<String>,
  },
};

action "viewPhoto" appliesTo {
  principal: User,
  resource: Photo,
  context: {
    "authenticated": Bool,
  }
};
```

Policies can be statically validated against a schema using the Cedar CLI, or the crate. They lend themselves to static analysis, and to being stored in a database. Fundamental properties of the Cedar policy language have been verified in Lean. Also see this summary on the Cedar security model.

In a first step, we would author and embed a Cedar schema and policies in the API, and start enforcing access to resources in the GraphQL API based using the Cedar engine behind the trait described in Section 4.1.1.

Since the policy language is easy to read and write, and our target audience is technically savvy, we could in the future allow organizations to define their own policies against our schema, and take them into account in the managed offering. In the Enterprise Platform, we could take the policies as a file path. The default policies will be available on GitHub, and customizing them would start with forking that repository.

Because the policies have a one to one mapping to JSON, we could build UI elements to customize parts of the policies, or all of them, and store the results in Postgres. The whole static validation logic for policies is in the open source cedar-policy crate, so these policies can be statically validated for type-correctness and applicability.

Since policy evaluation is a pure function, these policies are easy to test. We should aim to provide a good test suite with our base policies repository.

Audit logs are also an important feature. We should allow for them in the design, but the details are left to a separate RFC.

4.3. Machine to machine

For machine-to-machine communication, the existing implementation has access tokens. They are JWTs. They are long lived by default, can have an expiration time, and can be revoked. Access tokens are minted by the Grafbase API and enforced by the Grafbase API, as well as the Telemetry Sink and Object Storage services.

4.3.1. Access token security

We will keep long lived access tokens for third-party integrations and the Gateway. The Gateway needs long lived access tokens for reliability, so it does not depend on a token refresh endpoint on the API or another authorization server. We will keep the existing system, with only graph scopes, for Gateway access tokens, with scopes baked into the JWT.

We will also introduce access tokens with fine-grained permissions stored in Postgres, for access tokens that only need to communicate with the API. The fine-grained permissions will be built in the UI in the form of a (resource type, action) pair, with a wildcard ("*") or specific IDs for the resource. As you can see in grafbase/cedar-policies, the AccessToken entity has different properties from User, intentionally, to closely map to this way to assign fine-grained permissions.

Instead of the current symmetric encryption algorithm (HMAC), we will switch to asymmetric encryption. The algorithm will be Ed25519 (EdDSA), since it is fast and supported by AWS KMS.

4.3.2. Workflow identity

We will implement support for OIDC based workflow identity, with fine grained permissions, which will be defined with the same UI as fine grained permissions on access tokens.

Each organization will be able to configure which OIDC providers they trust, and map claims to fine-grained permissions. A full description of the feature is best left to a separate RFC.