Cedar, Rego, and OpenFGA Policy Languages

Comparative Language Security Assessment

August 21, 2024

Prepared for:
Amazon Web Services

Prepared by: **Ian Smith and Kelly Kaoudis**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor

Brooklyn, NY 11215

https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Analysis Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Threat modeling projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Table of Contents

Executive Summary

Engagement Overview

Amazon Web Services (AWS) engaged Trail of Bits to comparatively assess several authorization and access management policy languages including AWS' policy language: Cedar. The goals of this work were to identify broadly applicable threats to policy languages, to identify language features that partially or fully mitigate those threats, to identify existing features that exacerbate those threats, and to provide security recommendations to improve the general design of policy languages. We conducted threat modeling exercises identifying broadly applicable threat scenarios, aggregated these scenarios into threat groups, evaluated language and ecosystem mitigation maturity per group, and listed features that either mitigate or contribute to (worsen) each threat type. Refer to appendix B for maturity-level descriptions. We also discuss the potential impact of different deployment scenarios on appropriate policy language usage.

Observations and Impact

We identified sixteen general threat scenarios, and then grouped them by mitigation and attack surface similarity in order to more directly discuss the involved language features. We present the following high-level takeaways for each language with respect to our identified threat groupings:

Cedar's predictable evaluation model lacks side effects. This purity results in better ability to prevent denial of service attacks against policies by users (Threat Group 2). Formalized semantics, differential testing, and type safety provide mitigations against language misuse and mis-evaluation (Threat Group 1). Policy writers using Cedar should, however, take care when preparing data for ingestion. The policy writer must correctly perform normalization and build transitive relations (Threat Group 5).

Rego is an expressive policy language providing built-in support for a variety of common authorization-related data types and use cases (Threat Group 5). The flexible nature of the language, including the existence of side-effecting operations, presents security challenges for its deployers and policy writers, particularly in the areas of predicting performance and potential for language misuse or mis-evaluation (Threat Group 2 and Threat Group 1).

OpenFGA's modeling language provides a declarative interface for specifying ReBAC (relation-based) authorization constraints, which can simplify policy writing (Threat Group 1). OpenFGA's model management interface also includes mitigations for threats related to policy extension and replacement (Threat Group 6). The declarative style of models causes unpredictable performance, in some cases exacerbating risks related to denial of service (Threat Group 2). Attribute based authorization either relies on object names or expands the attack surface by including Google Common Expression Language (CEL) conditions (Threat Group 5).

How to Read this Report

In this section we provide suggestions regarding how to approach reading this document depending on the reader's role.

Language Designers

This report is primarily written for policy language designers to understand how the features of their language contribute to or detract from its potential for secure use. This means we suggest that policy language designers read all report sections. When reading our threat model, we recommend focusing on the trust zones, the overall system diagram, and the threat scenarios. These elements of the threat model are the basis of the threat groups and trust relationships that we used to compare language features.

When approaching the language security comparison, we suggest first reading the threat groups that are labeled with an overall mitigation maturity of moderate or weak. These threat groups represent areas where we believe the design of policy languages can be improved. For threat groups that are applicable for a given language design, the recommended mitigations presented in the summary table provide guidance on remediating these threats. Note that we refer to features that make it easier to use the language securely as *mitigating features*. We refer to features that increase attack surface and worsen a threat scenario as *contributory features*.

Application Developers

To make this work more accessible for developers working on different types of systems, we discuss several common policy language engine deployment strategies: multi-tenant, single-application, and single-host. Identifying which type of deployment a developer's policy language use case most closely matches will help the developer select the most relevant parts of the language security comparison. To determine how policy language features may impact the security of a given deployment, we recommend starting with the system diagram (which shows the multi-tenant model) and our other deployment scenarios, and then moving to the sections of the language security comparison deemed most relevant to the given deployment scenario. Note that our listings of contributory features for each threat group include misuse-prone language features that may need to be compensated for in the application or in the overall system design.

Security Engineers

We suggest security engineers cover the sections relevant for an application developer, but also pay additional attention to the threat model section and to all mitigations covered in this report. Our discussion of deployment scenarios combined with the threat model in this document may help identify gaps in the security design for systems that integrate or rely on the correct, secure use of a policy language.

Project Summary

Overview

During the project, we developed a threat-based comparative assessment framework for policy languages. Our threat-based approach allows for comparing policy languages in terms of language features that contribute to, or mitigate, each identified threat or threat group.

Our initial threat modeling exercises focused on a multi-tenancy scenario where an authorization service provider deploys a policy language evaluation engine that responds to authorization requests from multiple mutually distrustful applications. Our focus on the multi-tenant scenario during the first portion of this engagement was motivated by its generality. The trust boundary set of the multi-tenant deployment scenario broadly overlaps with the trust boundary sets of the single-application and embedded deployment scenarios we also consider later in this work. A multi-tenant threat model is a generalization of a single application threat model to multiple mutually distrustful applications that surfaces many threats that would not apply to the single-tenant scenarios. All identified threats can be found in table 1 in the Threat Scenarios section.

Following threat modeling, we analyzed each language's mitigation maturity for each identified threat. We established the maturity level by identifying contributory and mitigating features of the policy language under evaluation for each threat. Contributory features *decrease* the difficulty of exploiting a threat vector or *increase* the impact of exploitation. Mitigating features *increase* the difficulty of exploiting a threat vector or *decrease* the impact of exploitation. By comparing these two sets of features for a language and threat, we arrive at a mitigation maturity evaluation for the language with respect to the threat. The contributory and mitigating features, along with the final mitigation maturity assessment for each threat, can be found in tables 2, 3, and 4 in the Language Security Comparison section.

Finally, we grouped threats by overlapping contributory and mitigating features to identify similar threat groups. We aggregated mitigation maturity scores by threat group for each compared language by taking the average mitigation maturity of all threats in each threat group. Discussion of the identified threat groups, the current state of mitigations, and recommendations for future mitigation can be found in the Threat Mitigation Analysis section.

Scope

Threat Model

Our threat modeling exercises considered a generalized, ideal policy language system and enumerated potential threat scenarios that directly affect system components. Such a system consists of the language itself, any policy validation mechanism, the policy execution engine, sources of requests for access or authorization, and sources of policies, as broken down into components and trust zones below.

Language Security Comparison

Building on our threat model, our language-security assessment work compared the security of Cedar at version 3.2.0, OPA's policy language Rego at version 0.64.1, and OpenFGA's modeling language at version 1.5.3.

Out of Scope

Components of the OpenFGA and OPA servers (aside from the target languages) that could not be compared directly with components of the three policy languages themselves are broadly out of scope. Specific system deployments such as Amazon Verified Permissions that incorporate any of the policy language systems under comparison (Cedar, in the case of Verified Permissions) as a component and use it to produce access-control decisions are also out of scope.

Though we do mention select out-of-scope components as needed to contextualize our work, we also do not consider threats directly to the following components that may commonly be included in a broader system reliant on such a policy language:

- Any enforcement component that can be considered a Policy Enforcement Point (PEP), since the policy languages that we compare in this document do not directly include enforcement.
- Hosting/infrastructure providers, since the systems we compare can be deployed in several different configurations and can be deployed without requiring a particular infrastructure provider.
- Infrastructure-level components including any CI/CD pipeline; any proxying or reverse proxying component; any load-balancing mechanism; and the underlying virtualization layer or host on which any component in the model runs.
- Any public key infrastructure that could be in use for component identity creation and validation.
- Any third-party language extensions, policy modules, libraries, or functionality.

Finally, we make no assumptions about the types of users, clients, or client devices an application (that is, a policy language engine tenant) can have, and we consider user/client devices also out of scope.

Assumptions

Assumptions that may be useful to be aware of when reading this report include the following:

- Multiple application infrastructure trust zones (representing independent applications, also called “tenants” of the policy engine in this report) rely on the same policy decision point (PDP)/policy engine.
- Each application may have multiple, potentially distinct, mutually distrustful users.
- A user of a given application *may*, but need not, use or subscribe to other applications that are tenants of the same policy language system.
- Interstitial controls and enforced access policies may exist in any particular deployment that are not directly relevant to the policy language system itself.

Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive review of the following components, which may warrant further review:

- We could only partially evaluate OpenFGA’s CEL (Common Expression Language) conditions extension .
- We did not evaluate the correctness or scope of Cedar’s separate Lean formalism. We used this formalism as a supporting artifact for the correctness of the Rust Cedar implementation through AWS’ use of differential tests. The Rust Cedar implementation and its documentation were considered the final source of truth for Cedar language semantics.

Terminology

Where appropriate to contextualize our work, we leverage the vocabulary of RFC 2904: AAA Authorization Framework. We understand this terminology, while non-normative, to be historically common in the access control community, particularly among those who are familiar with XACML.

We label locations and components that take on roles and conventions outlined in RFC 2904, including: policy decision points (PDPs), policy enforcement points (PEPs), policy information points (PIPs), policy retrieval points (PRPs), and policy repositories. We label components when they fulfill these roles, but maintain distinct nomenclature since components in our model may fulfill multiple roles, and multiple components may fulfill the same role. Where illustrative, we also draw relationships between any distinct terminology we use and anything overlapping that is defined in RFC 2904.

It should also be noted that we model specific instances of trust relationships that are covered by, or tangential to, RFC 2904, but do not directly model or recycle all trust relationships or deployment scenarios covered in RFC 2904. See Data Flow - Deployment

Scenarios for a brief, comparative description of the deployment scenarios and different sets of trust relationships that are covered in this document.

Threat Actors

In the following table, we define the types of *actors* that interact with the system and consequently can impact the security of the system. An actor is an entity, or a distinct set of privileges that enable an entity to perform access-controlled actions. Actors may also be impacted by, or induced to undertake, an attack. We will reference these actors to help contextualize the data flows, trust zones, and threat scenarios later in this document. Establishing the types of actors that could threaten the system is also useful in determining which protections, if any, are necessary to mitigate or remediate vulnerabilities.

Actor	**Description**
User	A User is positioned externally to the applications that use the policy engine to authenticate and authorize requests. As described in RFC 2904, a User who wants access to a service or resource interacts with an application that mediates and submits policy requests to the policy engine on their behalf. A User may be an individual interacting with the application via some out-of-scope client, or may be a different service or application.
Application Maintainer	The application maintainer creates and hosts some application that interacts with the policy engine on behalf of the application's users. The application provides policies, authorization data, and other contextual information that informs the generated policy decision.
Policy Retrieval Agent	The Policy Retrieval Agent role, analogous to the RFC 2904 Policy Retrieval Point (PRP) agent definition, is a privilege set that enables its member(s) to retrieve policies from one or more policy repositories in the system. A Policy Retrieval Agent could be an individual running a policy language engine or policy validator application on the command line and manually providing policy locations. This privilege set could also apply to a policy engine retrieving policies from the policy repository, or to an application retrieving policies and providing them to the policy evaluation engine along with an access control request.
Identity Provider Updater	The identity provider updater has privileges to create new entries in the identity provider, or to modify existing entries (such as adding an LDAP user account to a role). The identity provider updater may additionally have the ability to remove existing entries.

Authorization Provider	The authorization provider hosts the policy-related components that handle policies, requests, and authorization information from various applications and returns authorization decisions. Hosted policy engines often store policies and authorization data for multiple applications in the corresponding policy repository.
External Attacker	An external attacker is positioned on the broader internet. This actor can eavesdrop on and potentially modify legitimate traffic to the policy engine or to a given application that crosses the external network. An external attacker who compromises a legitimate User's credentials can submit maliciously crafted requests.
Internal Attacker	An internal attacker is positioned in the Provider zone where policies are created, stored, and deployed. This actor can eavesdrop on and potentially modify legitimate traffic to the policy engine that crosses this internal zone. Furthermore, this actor can potentially modify legitimate traffic to and from a remote identity provider, or to and from the policy validator.

Components and Trust Zones

This section presents the general trust zones and components modeled in this document. Since we intend this threat model to apply generally to policy languages for identity and access management, some details that might more closely adhere to one particular system over another are abstracted away.

Note that a brief discussion of different component deployment scenarios, as well as how trust boundary positioning and relevant threats change by deployment, can also be found later in this document under Data Flow.

Trust Zones

A *trust boundary* is a location between connecting components in the system where interstitial security controls and access policies are, or should be, enforced. A *trust zone* is a grouping of components that share the same trust boundaries and criticality at the modeled level.

Trust zones are delineated by the security controls that enforce the differing levels of trust within each zone. Therefore, it is necessary to ensure that data cannot move between trust zones without first satisfying the intended trust requirements of its destination.

We divide our policy language model system into five types of trust zones: User, Application Infrastructure, Provider, Entity Information, and Telemetry.

Application Infrastructure

The application infrastructure trust zone encompasses the application itself, including the infrastructure used to host its data and perform computations. In this model, the maintainer trusts the host of their application. We assume that, in the context of a multi-tenant authorization provider, there may be *n* application infrastructure trust zones (representing independent applications), and *m* multiple users of each application. This trust zone is roughly analogous to the User Home Organization entity that RFC 2904 defines.

User

The User trust zone represents the trust barrier and corresponding need for authentication and authorization between users and the applications they interact with. While RFC 2904 does not clearly state whether one or many users are modeled and does not clearly make assumptions about allowable user-to-user interactions (or lack thereof), in our model any application may have multiple users, and each user operates a distinct client/device.

Provider

The provider trust zone includes the infrastructure an authorization provider operates using a policy language in order to store, update, and evaluate policies on behalf of *n* applications, each of which have *m* users.

Entity Information

The entity information trust zone provides an externalized and abstract source of entity and identity information used by applications and the provider trust zone as context when evaluating policies. This trust zone is analogous to an identity service or store that may be operated by a distinct third-party vendor, or that may be the responsibility of a distinct internal team upon whose services the authorization provider team have a dependency.

Telemetry

The telemetry trust zone captures a variety of logging services that can be used both by the application and policy engine, in order to provide logs for the use of an application maintainer. We use a single trust zone to represent all telemetry to simplify the non-core model elements.

While having a single telemetry zone most closely models use cases where both the application infrastructure maintainer teams and the authorization provider team share a telemetry sink, the telemetry sinks that the applications and the provider zone each connect to could also in practice be entirely separate, mutually distrustful services.

Components

The following table describes the *components* that comprise the modeled policy language system and maps each component to the generally modeled trust zone in which it is located. For a visual representation of this modeled architecture deployed with multiple tenants, see System Diagram. All components described in the following table are in scope. We explored the implications of some threats that involve out-of-scope components and directly affect in-scope components, but we did not consider threats to out-of-scope components themselves. For a brief overview of *out*-of-scope components that relate to the policy language system, see Scope.

Component	Trust Zone	Description
User Device	User	The User Device sends a request to an application to access or modify some resource.
Application	Application	An Application outsources authorization decisions for

	Infrastructure	User requests to the policy engine and, if authorized, performs the requested action(s) on behalf of the User in question. We model the application as a possible Policy Retrieval Agent (roughly as described in section 4.4: Distributed Policy of RFC 2904) since some policy language systems allow for applications/requesters that can provide policies to the policy engine along with the access control request.
Application Policy Store	Application Infrastructure	The Application Policy Store holds policies on behalf of the individual application to which it belongs. These policies will ultimately be used to inform the policy engine's authorization decisions. Policy Retrieval Agents (analogous to the agent description in section 4.1: Policy Retrieval of RFC 2904) can interact with a given Application Policy Store. The policy engine—one possible Policy Retrieval Agent—can source policies from the individual application's policy store, or the application (another possible Policy Retrieval Agent) can provide the engine with relevant policies for evaluation over a given access control request. We model this component as distinct from the more general Policy Repository, since the Policy Repository stores policies related to an entire organization rather than a single application.
Log Store	Telemetry	Each application—as well as the policy engine —logs to a remote log store operated by an external infrastructure provider. Facilities for access to the stored logs may be unauthenticated, may be authenticated and authorized using the policy engine, or may be authenticated and authorized through an ancillary system.
Policy Repository	Provider	The Policy Repository stores policy definitions that can apply to multiple, distinct, mutually distrustful applications and their users. The policy execution engine accesses the policy repository at runtime prior

		to policy evaluation, or just in time after receiving a request that needs evaluation. As described in RFC 2904, policy definitions are maintained and stored in a policy repository by (or on behalf of) the organization that requires them. Policy Retrieval Agents have privileges to interact with policy repositor(ies).
Policy Execution Engine	Provider	The Policy Execution Engine is a single language engine that evaluates policies over a given request and returns a single decision about that request back to its originator. This engine is the only Policy Decision Point (PDP) in this model. The PDP is described as the location at which policy evaluation occurs in 4.4. Distributed Policy of RFC 2904. The engine can receive authorization requests from multiple applications. Authorization decisions the engine makes can involve policies sourced from the application's policy store in addition to or instead of information sourced from the policy repository, and are based on relevant information available from the identity provider. We also model the engine as one possible Policy Retrieval Agent (Policy Retrieval Point) as described in section 4.1: Policy Retrieval of RFC 2904, since some policy language systems require the execution engine to be backed by, and have access to, a generalized policy repository. This means that the engine itself is another potential policy repository, if it implements some form of local policy caching or uses an embedded policy store rather than leveraging an external data store as its generalized policy repository.
Policy Validator	Provider	The Policy Validator receives candidate policies from the application. The validator returns to the application a yes or no validation decision depending on whether the policy can be validated against acceptable constructs in the policy language, and against any additional rules or relationships the application or request source provides along with the

		policy. Note that policy *validation* is a distinct procedure from policy evaluation. Validation may or may not involve checking the policy in question against sample request data. We also model the validator as one possible Policy Retrieval Agent (Policy Retrieval Point) as described in section 4.1: Policy Retrieval of RFC 2904.
Identity Provider	Provider	The identity provider provides a secure record of a number of entities such as principals and their groupings and roles, as well as what privileges and capabilities are associated with these groups and roles. The policy execution engine may query the identity provider, or in some cases, the application may query the identity provider and then query the policy execution engine.

System Diagram

In the below system diagram, we depict the generally modeled connections between the components and trust zones of the idealized policy language system. The direction of each connection between components indicates the connection's originator. Dotted red lines indicate trust boundaries separating the different trust zones. We also use coloration and the included diagram key to map between the concepts covered in this threat model, and the terminology of RFC 2904. Note that there are both multiple, distinct User zones, and multiple, distinct Application Infrastructure zones modeled. Depending on design choices, policies can be sourced from an application-specific policy store, or from a generalized policy repository store, or from an embedded data store or cache within the engine itself.

The Components and Trust Zones section of this document above provides more detail on the different trust zones and components pictured, and the Data Flow section below provides detail on the connections between components that cross trust boundaries.

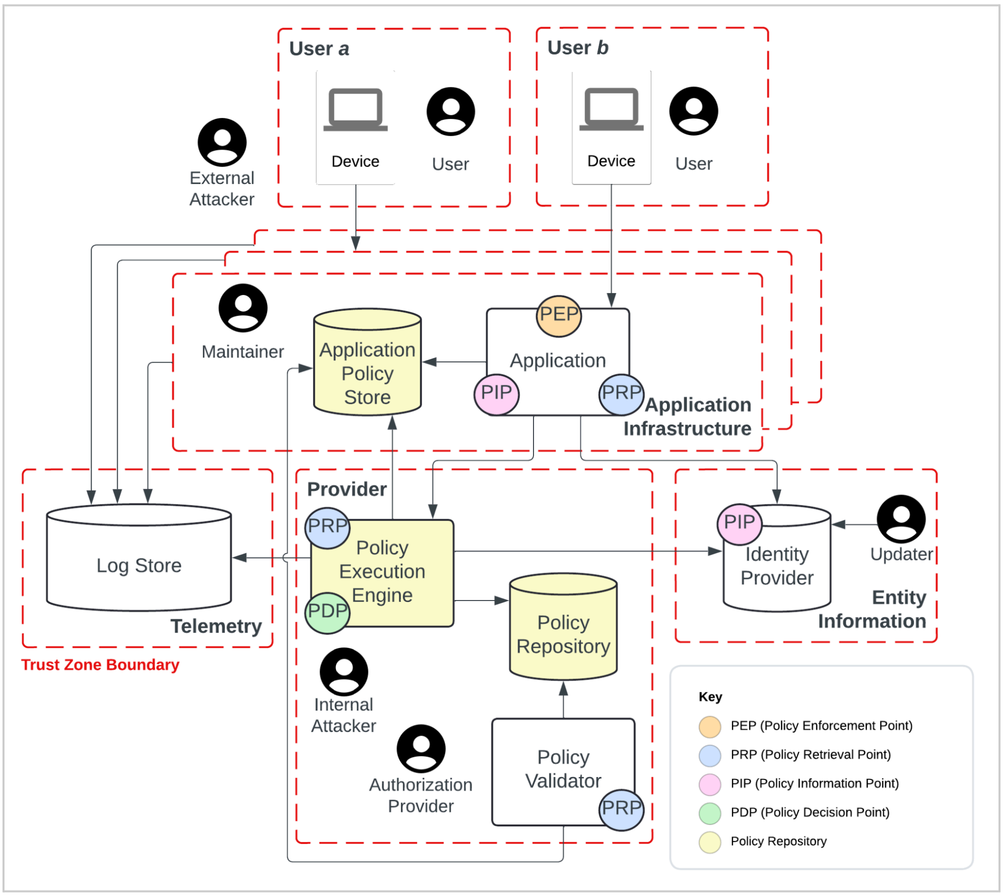

Figure 1: Connections across trust zones within the multi-tenant deployment scenario of the modeled policy language system

Data Flow

At a design level, trust zones are delineated by the security controls that enforce the differing levels of trust within each zone. Therefore, it is necessary to ensure that data cannot cross the trust boundaries between zones without first satisfying the intended trust requirements of its destination. We enumerate such connections between trust zones below.

Type of Data	***Source Zone***	***Destination Zone***	***Description***
User Request	User	Application Infrastructure	A user submits a request using a given principal credential to access a given resource owned by the target application.
Application Request	Application Infrastructure	Provider	An application supplies a formatted request to the authorization provider's infrastructure. The application request contains a request type, and principals or resources. An application request may also contain contextual data used for evaluating the request.
Policy	Application Infrastructure	Provider	An application or application maintainer transmits policies to the provider to be evaluated when servicing authorization requests. These policies may be stored in the application's individual policy store and then retrieved for evaluation, or evaluated in-place by the policy engine during an ad-hoc query.
Authorization Model	Application Infrastructure	Provider	An application or application maintainer can upload an authorization model against which policies or tuples are validated during uploading, modification, or in-place evaluation.

Entity Information	Application Infrastructure	Provider	Entity information defining a document, identity information, or tuple state under which a policy is run may be transmitted by the application to the provider.
Entity Information	Identity Provider	Application Infrastructure	The application may source entity information from the Identity Provider when providing entity information to the provider for policy evaluation.
Entity Information	Identity Provider	Provider	An Identity Provider may provide entity information directly to the execution engine either prior to or during policy evaluation.
Authentication Request Response	Provider	Application Infrastructure	The provider creates a response to an authentication request based on the context and policy. This response can include whether the request is authenticated and additional error information.
Context Values	Provider	Application Infrastructure	Some policy languages allow for extracting context metadata, such as the transitive closure of relations or document values. This additional information can be returned to the application along with the request response.
Application Telemetry	Application Infrastructure	Telemetry	The application transmits logs to the log store. Information logged may be related to, or may include, authentication and authorization decisions.

Policy Execution Engine Telemetry	Provider	Infrastructure	The policy execution engine logs to the log store. Information logged may be related to, or may include, the availability of the identity provider, which application(s) and users(s) have requested authentication and authorization, and authentication and authorization decisions.
Configuration Information	Application Infrastructure	Provider	The application may provide configuration data, such as logging destinations, data sources, data store destinations, and authentication information, to the provider.

Deployment Scenarios

The above system diagram and data flow connections table cover a multi-tenant scenario, but they also partially apply to other possible deployments: namely, a single-tenant (app) scenario, and a single-host (embedded PDP) scenario. Here, we note distinguishing features of each deployment scenario and briefly discuss how changing or removing trust boundaries from our generalized multi-tenant model changes the potential attack vectors that should be considered.

Multi-Tenant

A single PDP (policy language engine) serves as the access control decision source for *n* multiple, mutually distrustful applications, each of which may have *m* multiple, mutually distrustful users. This PDP may either be a component of a larger system specifically geared toward one type of access control decisions, or may support multiple, disjointed types of policies across a variety of use cases.

The lack of trust between applications in this deployment scenario requires boundaries between applications, and also a boundary between the applications and the policy execution engine as in the above system diagram. An internal attacker within the provider zone may have direct access to the general policy repository, for example, but not to applications, which may act as their own PEPs or may outsource policy enforcement to another out-of-scope service.

The attacker may interact with the policy language system proxied by any of the *n* applications that depend on the engine for access control. Request proxying through further infrastructure may additionally be required between the various trust zones and may enable further types of attacks depending on the deployment, though any such proxy component is out of scope for this assessment.

Single Application

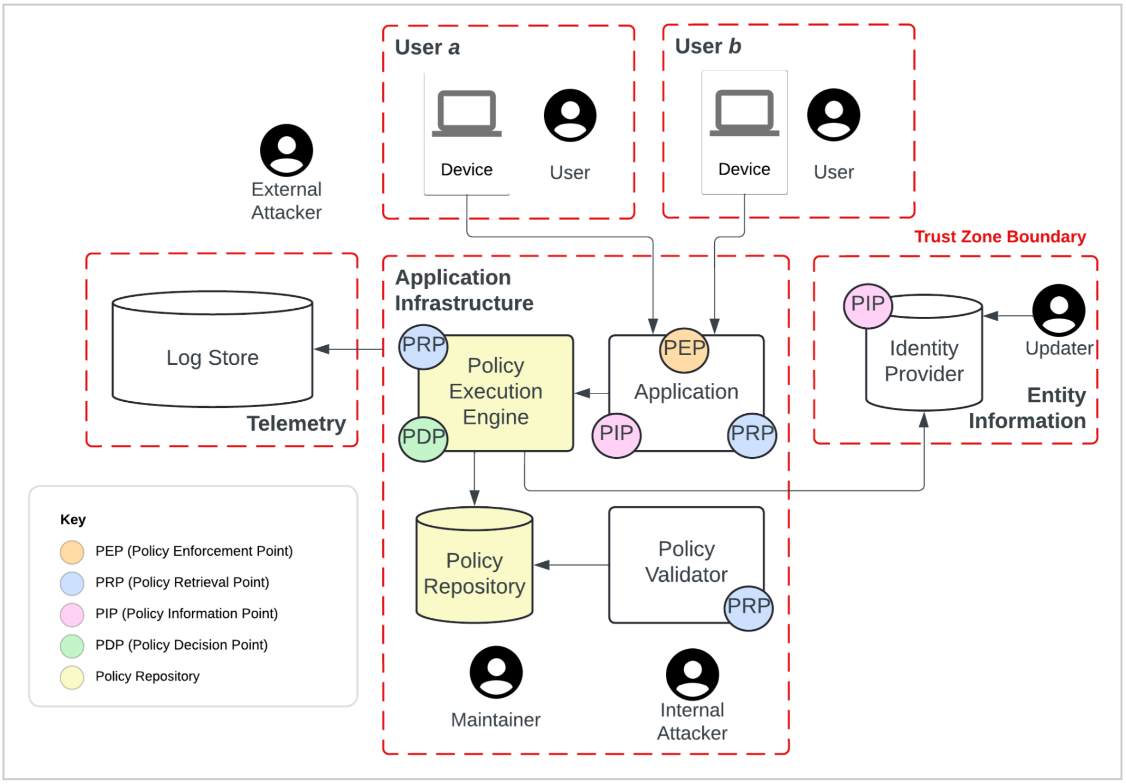

Figure 2: Connections between trust zones within a modeled single-application policy language system

In a single application model, a single, dedicated policy execution engine serves as the access control decision source (PDP) for a single application with multiple, mutually distrustful users. The engine, application, and any other supporting infrastructure such as the telemetry sink could be, for example, distinct networked services co-located within the same cloud provider account or on-premises deployment. There could be, but potentially need not be, interstitial access controls between these services, leading to our placement of them within the same trust zone in this model.

Unlike in the multi-tenant scenario, in this single-application deployment the engine trusts the (sole) connecting application. In the case that the engine *does not* trust the connecting application and requires application authentication and/or authorization on inbound connections, the deployment should be considered an instance of the *n* tenant model above with *n*=1, since the trust boundaries would then be the same. We also collapse the policy repository and the application-specific policy store into a single store in this deployment scenario.

The same entity or team may hold both authorization provider and application infrastructure maintainer roles in this model. For example, a single team may run an application and also run their own corresponding policy engine. The application may support a wide variety of distinct users.

Single Host

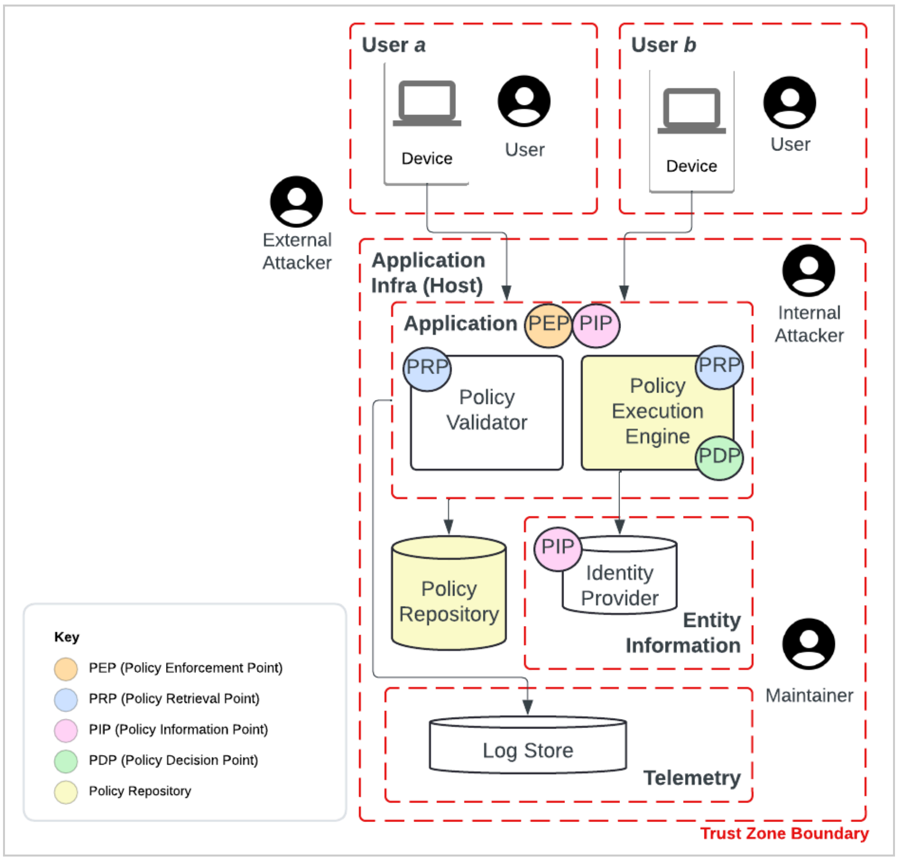

Figure 3: Connections between trust zones within a modeled single-host policy language system

An application embeds a library version of the policy engine in question. Now, the connections between the application and the policy engine are either IPC (inter-process communication) between processes on the host, or direct policy engine library API calls, depending on whether the policy engine library runs in a distinct process on the same host as the application, or is simply another component directly imported into the application. The application may additionally call the policy validator in the same fashion (e.g., as yet another library).

The primary differentiators between the single-host and the single-application models are the types of interstitial component mediation and isolation mechanisms that an attacker is *obliged* to navigate in order to exploit the system. As pictured above, the Application Infrastructure zone becomes a single host. We also reframe the telemetry sink and the identity provider as services running in additional processes, perhaps under different user accounts, on the same host.

Threat Scenarios

The following table describes possible threat scenarios and related actors and components given the modeled invariants of a policy language system.

Threat Identifier	Actor	Scenario	Description
TOB-T1	User, External Attacker[1] **1.** An external attacker may also be able to leverage policy evaluation errors if they can use stolen credentials or use an authorization bypass in order to cause the application to evaluate a policy.	Bypasses Policy Requirements to Access Application Resources	A User takes advantage of an error in the policy, of an error in its context preparation, or of policy misevaluation to bypass an application's authentication procedures and gain inappropriate access to application resources.
TOB-T2	Application Maintainer, User	Application Maintainer Leaks Authorization Decisions in Logs	An application maintainer logs sensitive information, such as the principals, groups, roles, or actions involved in authentication or authorization decisions. Consequently, an internal attacker who gains access to the logs service is able to learn the needed privileges for access to sensitive internal resources or retrieve sensitive information based on actions taken by a principal.
TOB-T3	User, External Attacker[1] **1.** An external attacker may also be able to cause a denial of service if they can use stolen credentials or use an	User Induces a Denial of Service on the Policy Engine	A user induces a denial of service on the policy execution engine that the application amplifies by inducing repeated requests, creating significant contextual information, or both. The induced denial of service could be a consequence of the language and evaluation implementation; for example, recursion over crafted

	authorization bypass in order to cause the application to evaluate a policy.		request elements could lead to denial of service through requested infinite input element expansion.
TOB-T4	Application Maintainer	Application Maintainer Induces a Denial of Service	An application creates a denial of service for other applications by creating excessive computation on the policy engine or on the policy validator through repeated requests, or through creating a policy that consumes significant resources to execute, or through creating an overload of contextual information, or through a combination of these factors.
TOB-T5	Application Maintainer	Application Maintainer Inappropriately Includes External Resources	Suppose the system supports the inclusion of external code or entities in policies. Then, an application maintainer writes a policy that references a resource outside of the application maintainer’s control, such as a resource available from the public internet. An external attacker later discovers and co-opts this external resource. For example, such external resource co-opting could occur through DNS hijacking, or even through malicious commits within the included external package, if it is open source.
TOB-T6	Application Maintainer, Authorization Provider, Internal Attacker	Authorization Provider Fails to Disallow Inappropriate Internal Resource	Suppose the system supports the inclusion of externally provided code or entities in policies and fails to disallow referencing internal locations. Then, an internal attacker or a compromised

		Referencing	application maintainer could write a policy that reads from internal system resources inappropriately, such as `/etc/shadow` or other host configuration files.
TOB-T7	Authorization Provider, Application Maintainer, User, External Attacker	Unexpected Encoding Schemes Leveraged for Access to Inappropriate Internal Resources	Suppose the authorization provider fails to uniformly and correctly handle alternative and/or historical IPv4 or IPv6 or Unicode encodings, and/or Unicode control characters, in policies. Then, a policy written by the application maintainer could, for example, allow access to unexpected internal resources, or could allow the external attacker the ability to craft requests to the application that cause the application to make further inappropriate internal requests on the attacker's behalf. Alternatively, if different encodings are not uniformly handled, an application maintainer could intentionally use different but visually similar Unicode encodings to indicate different resources, or a compromised application maintainer could use homoglyph Unicode characters to create policies that validate but are not sufficiently restrictive.
TOB-T8	Application Maintainer	Application Maintainer Achieves Data Exfiltration through the	An application maintainer takes advantage of a memory corruption, serialization, or other application security flaw to gain undue visibility into provider or

		Policy Interpreter	application data that is not owned by the offending application.
TOB-T9	Application Maintainer	Application Maintainer Manipulates Policy Request Responses through the Policy Interpreter	An application maintainer takes advantage of a memory corruption, serialization, or other application security flaw to manipulate interpreter behavior or context information in order to cause incorrect policy evaluation for another application.
TOB-T10	Application Maintainer	Application Maintainer Manipulates Application Data Through Policy Execution	An application maintainer uses side-effecting features of the policy language to manipulate context data for another application.
TOB-T11	Application Maintainer	Application Maintainer Reads Application Data through Execution of the Policy	An application maintainer uses features of the policy language to read context data that belongs to another application. The policy engine may then return this context data back to the application as part of its response if it supports arbitrary response data, or may write this context data out to another sink such as the logs store.
TOB-T12	Application Maintainer, Internal Attacker	Internal Attacker or Compromised Application Owner Uploads Purposely Ineffective Policy	An internal attacker gains access to change policies that the policy execution engine evaluates for an internal resource and consequently uploads an overriding policy that allows them access to that internal resource.

TOB-T13	Application Maintainer, Internal Attacker	Internal Attacker or Compromised Application Maintainer Adds Malicious Policies	Suppose policy evaluation order matters for the final policy execution engine decision, and multiple policies can factor into the same authorization decision. Then, an internal attacker could add malicious policies that override the intended authorization restrictions at runtime. Alternatively, a benign but uninformed member of the application maintainer team could accidentally add a new policy that negates a previously added but necessary policy for some authorization decisions, leaving previously gated resources unguarded.
TOB-T14	Application Maintainer, Internal Attacker	Internal Attacker or Compromised Application Maintainer Modifies Policy Dependency Modules	An internal attacker gains access to an application's policy repository to change application-specific modules that multiple production policies of the application import. The attacker then overwrites a module in a way that breaks production access for legitimate users, or that grants the attacker inappropriate access to internal resources without needing to directly change the policies that import the module and gate resource access.
TOB-T15	Authorization Provider, Internal Attacker	Internal Attacker or Compromised Authorization Provider Modifies	An internal attacker gains access to the policy engine or to its storage, and consequently modifies built-in language

		Language Builtins	functionality to break production access for legitimate users, or to grant themselves inappropriate access to internal resources without needing to directly change the policies that gate access.
TOB-T16	Authorization Provider, Internal Attacker, Application Maintainer	Internal Attacker Intercepts and Modifies Policy Engine Responses	Suppose the policy engine is deployed to the application with improper protections for the communication channel (either a network connection or library interface). An internal attacker with access to the unprotected channel may be able to replace or modify policy engine responses to the application to achieve unintended authorization behavior.

Table 1: Threat scenarios

Language Security Comparison

In this section, we compare the features of the three policy languages under evaluation with respect to security: Rego, OpenFGA's Authorization Modeling Language, and Cedar. The threat model outlined in the previous sections scopes and informs this comparison.

We group the threat model's more granular threat scenarios into closely related sets by attack surface and mitigation similarity that we term *threat groups*. For each threat group, we first identify each individual language's set of contributory and mitigating features. We refer to features that make it easier to use the language in question securely as *mitigating* features. *Contributory* features, on the other hand, increase attack surface and make it more difficult to use the language securely.

We then evaluate the maturity of each language with regard to each threat and threat group on a scale of Weak to Strong; refer to appendix B for descriptions of these maturity levels. To get the overall mitigation maturity for a particular threat group, we take the maturity average over all evaluated languages and round down to the nearest maturity level. Finally, we discuss how the deployment scenarios outlined in our threat modeling exercises relate to the threat group in question.

Since a reasonably secure policy language engine deployment requires a combination of mitigations at the language level and at the system level, we include deployment-level threats in Group 7 and in deployment discussions. Beyond this threat group, we do not discuss threats that cannot be reasonably mapped to or addressed by language features. Such threats are primarily contributed to or mitigated by features of a particular deployment. We provide the same comparison organized by threat rather than by language in appendix A.

Threat Group 1: Language Usage and Authorization Logic Errors

Group 1 consists of threat scenarios in which an application maintainer writes a policy that does not sufficiently gate access to the resources that they intend to protect. This outcome can result from a mismatch between policy-writer expectations and the language interpreter's behavior.

Language Features and Maturity Evaluation

Language-level mitigations for this threat group include mitigations that decrease the possibility of misinterpretation, such as deny-by-default behavior, and analysis and testing tools. A simple authorization model also decreases the likelihood of policy misinterpretation. We found the overall ecosystem maturity for this threat group to be moderate. We also note there is a tradeoff between supporting a wide variety of authorization use cases and maintaining a simple and well-specified authorization model.

Threat Scenarios	Mitigation Maturity Summary	Recommended Mitigations
TOB-T1	Cedar: Moderate Rego: Weak OpenFGA: Moderate *Overall: Moderate*	• Analysis tools • Type checking • Testing frameworks • Semantic specification • Default-deny behavior

We break down the above summary by policy language and compare features that weaken as well as improve the language with respect to Threat Group 1 below.

Cedar

Threat	Contributory Features	Mitigating Features	Maturity Evaluation
TOB-T1	• Runtime errors result in skipped `forbid` policies • Runtime errors created by arithmetic operators (i.e., overflow/underflow)	• SMT Policy Analysis • Static Type Checking • `forbid` overrides `permit` • Deny by default • Explicitly defined	*Moderate*: The language provides a strong formal backing for its default `forbid` behavior, and there exist powerful analysis tools for users to confirm policy and language properties.

	and missing entities are not modeled by the validator	`permit` or `forbid` output • Formal semantics • Differential testing of evaluator against formal model • Style guide and best practices	Differential testing provides evidence of implementation equivalence with the semantic model. Cedar does not provide a dynamic unit testing framework for policy writers. Runtime errors can potentially result in a policy decision of `permit` that is annotated by errors.

Rego

Threat	Contributory Features	Mitigating Features	Maturity Evaluation
TOB-T1	• Flexible document model • Future keyword extensions leads to diverging language versions • Improperly handled, undefined values in deny rules lead to undefined result • Negation swallows undefined errors • Non-explicit forbid/allow output • Side-effecting and non-deterministic	• Policy testing framework • Default undefined decision • JSON schema checking • Style guide and best practices	*Weak*: The language provides schema checking, but propagates undefined results through intersecting rules which could cause undefined-by-default policy evaluation behavior. The testing framework allows policy writers to unit test their policies. A combination of features that allow for ignoring runtime errors (the `not` operator), weak document schema enforcement, and the ability to create complex, custom non-Boolean output formats increases

	built-ins		the possibility of writing unintentionally misconstructed policies.

OpenFGA Modeling Language

Threat	Contributory Features	Mitigating Features	Maturity Evaluation
TOB-T1	• Arbitrary relation queries • Non-explicit forbid/allow output • Limited to directly modeling ReBAC • CEL conditions add expressivity to write ABAC policies, but increase parsing and evaluation complexity • OpenFGA policy writers can unintentionally create CEL conditions that always error when the minimum cost estimate exceeds the maximum bound	• Model testing framework • Model specifies allowable relations • Style guide and best practices	*Moderate*: OpenFGA has by-default limited expressive power that presents a simple authorization model through relations. Relations are testable and fit an analyzable relational model. Possible relations can be visualized and analyzed by the policy writer. The limited expressive power of OpenFGA may require policy writers to perform further checks outside of OpenFGA or to extend the model with custom logic that may not be tested sufficiently and is subject to errors. The lack of negative tests or a defined formal semantic model increases the risk of policy misevaluation, particularly when introducing caching.

Deployment Scenarios

We briefly map Threat Group 1 to each of the three deployment scenarios under which we considered the three policy language engines.

An external attacker may not need to gain legitimate User credentials, or may only need to gain user credentials that were originally intended by the application maintainer team to have a lower level of privilege, in order to bypass an inappropriately scoped access control policy. For all three of the following deployment scenarios, this threat group involves some policy flaw allowing a naive or compromised user (or an attacker) to gain an inappropriate level of privilege or resource access that should be denied by the policy.

Multi-Tenant

The likelihood of an overly permissive policy increases as the number of tenants increase, despite any provided optional-to-use policy testing and validation tools, in a multi-tenant (versus a single-tenant) deployment. However, in this model the policy engine enforces access control for all inbound requests to it. Thus, the privileges and access to the components of the policy language system that an attacker could maximally gain through exploiting such a policy may be less in a multi-tenant deployment, when compared to those of a single-tenant deployment.

Single Application

Since there is no trust boundary, there is a lack of interstitial controls or access policy enforcement between the application and the policy engine in this deployment scenario. We hypothesize that the level of system access that an attacker who is able to bypass a policy can maintain (e.g., through editing or nullifying that policy completely) could extend further from the perspective of the policy engine. This means that an attacker who bypasses some access control policy could potentially more easily gain access to the policy engine, and obtain the ability to create and modify any policy of their choice, than in the multi-tenant scenario above.

Single Host

An external attacker who can bypass an inadequate policy in the case where the PDP/policy engine runs as an embedded library within a binary may then gain the ability to directly attack the running application that embeds the policy engine.

Threat Group 2: Denial of Service or Unexpected Performance Effects

Group 2 consists of threat scenarios where an actor induces denial of service for legitimate policy engine users through manipulating requests, the request environment, the policy itself, or some combination of these. Declarative authorization languages shift control over performance away from policy writers, creating the possibility for unexpected performance issues.

Language Features and Maturity Evaluation

The evaluated policy and modeling languages do not currently prevent an attacker with sufficient access (or a naive user) from creating expensive policies. This capability can be limited by externalizing relation building from the policy language itself (like in Cedar), but as expressive power gets added back to the language (e.g., with set operations), these features introduce performance risks. The evaluated policy languages are particularly weak to scenarios where an attacker controls the policy itself. All examined policy languages attempt to guarantee termination by restricting recursive rules. However, termination is a relatively weak performance guarantee.

Both Cedar and Rego provide predictable time complexity for linear fragments of the language, but an attacker controlling the policy can take advantage of expensive operations that depend on controlled inputs to increase runtime costs of a policy. In Cedar, the additional cost is bound to a worst case $O(N^2)$ for set operations where N is the number of elements, and $O(N^2)$ where N is the depth of the entity store for policy slicing. Rego's non-deterministic operations introduce further complications by ceding control of a built-in's runtime to some external server. Finally, OpenFGA relations are highly declarative, leading to operations that can have large and unexpected performance costs (e.g., exclusion operators require a global check for each object when performing a list object query).

Threat Scenarios	**Mitigation Maturity Summary**	**Recommended Mitigations**
TOB-T3, TOB-T4	Cedar: Moderate Rego: Weak OpenFGA: Weak *Overall: Weak*	• Clearly and publicly document the worst-case performance for language operations • Avoid non-deterministic language operations • Simplify evaluation in-order to provide predictable performance • Guarantee termination

We break down the above summary by policy language and compare features that weaken as well as improve the language with respect to Threat Group 2 below.

Cedar

Threat	Contributory Features	Mitigating Features	Maturity Evaluation
TOB-T3	• Worst-case performance of policy slicing is multiplicative in the depth of the entity store hierarchy • Set operations are worst case quadratic in the size of sets in the entity store • Performing policy principal or resource bounding in when clauses prevents policy slicing on the basis of those bounds	• Termination proof for the Lean formalization • Policy slicing reduces performance impact of large policy stores • Pure policy evaluation by default	*Strong*: Cedar policies are generally efficient to evaluate and lack side effects that could run for an unbounded timeframe. An attacker with a high degree of control of the entity hierarchy can induce quadratic work during policy slicing (due to quadratic policy keys) or policy evaluation (if the policy uses set operators).
TOB-T4	• All features from TOB-T3	• All features from TOB-T3	*Moderate*: The threats of application-provided policies and entity information at runtime both contribute to the attack surface of Cedar. A malicious or naive application maintainer could create large policy stores with many template instantiations and a large entity hierarchy to produce expensive requests.

Rego

Threat	Contributory Features	Mitigating Features	Maturity Evaluation
TOB-T3	• HTTP built-ins have unpredictable runtime determined by the endpoint, up to a provided timeout. • Net built-in `lookup_ip_addr` has an unpredictable runtime, up to the timeouts of underlying resolvers • Implicit indexing makes policy performance potentially unpredictable	• Default timeout for HTTP built-ins • Capabilities restrict built-ins to an allowlist • When policies are restricted to the linear language fragment, policies will run with worst case linear performance • Indexed policies transform some nested comprehensions to linear time • Policy profiling tools	*Moderate*: Using only the linear time language fragment of Rego avoids unexpected time intensive evaluation of user-generated queries. Capability restrictions can be used to prevent the usage of potentially expensive builtins. However, the language's built-ins can have unpredictable runtime characteristics that may be attacker-controlled and there is not a first-party solution for restricting language features to the linear fragment.
TOB-T4	• All features from TOB-T3 • Regex built-ins combined with control of the regex could lead to catastrophic backtracking	• All features from TOB-T3 • Regex implementation uses Thompson regex matching to be O(mn) in the size of the regex and input string	*Weak*: The threats of application-provided policies and documents at runtime both contribute to the attack surface of Rego. An attacker can leverage Rego's network-request-related built-in functionality to drastically increase policy evaluation time and resource consumption.

OpenFGA Modeling Language

Threat	Contributory Features	Mitigating Features	Maturity Evaluation
TOB-T3	● Persistent storage on writes increases the potential performance impact of tuple modifications ● List object requests can incur unexpected high overhead when evaluating a complex model	● Pure model evaluation by default ● Subproblem cache may reduce duplicate work in short timeframes	*Moderate*: Models and relations lack side effects and should be guaranteed to terminate on requests. However, evaluation can have unpredictable performance, especially when the model under evaluation contains intersection and exclusion operators.
TOB-T4	● All features from TOB-T3 ● Intersection relations in combination with a list objects call allow a maintainer controlling the model to create expensive calls ● Regex CEL conditions combined with malicious control of the regex could lead to catastrophic backtracking ● CEL macros in conditions combined with malicious macro expansion control could lead to infinite	● All features from TOB-T3 ● Global timeout settings for requests ● Regex implementation uses Thompson regex matching to be O(mn) in the size of the regex and input string ● OpenFGA sets a default maximum cost limit for CEL conditions ● CEL sets (tested) internal expression recursion and maximum recursion depth limits	*Weak*: The threat of application-provided models contributes to the attack surface of OpenFGA. Current versions of OpenFGA allow for easily constructing list requests that exhaust the global timeout for requests by using exclusion operators.

	macro expansion		🟥

Deployment Scenario Applicability

We briefly map Threat Group 2 to each of the three deployment scenarios under which we considered the three policy language engines.

Multi-Tenant

An external attacker or compromised user who captures and replays any request traffic against the policy execution engine or the policy validator, or an internal attacker who creates maliciously expensive policies, could negatively affect policy engine service levels for applications and users beyond the boundaries of any one policy engine tenant application.

Single Application

The difficulty of exploiting this threat is increased in the single application scenario. An external attacker attempting to create a denial of service is limited primarily to the privileges of an application user. Since the policy engine trusts the application, attackers attempting to deny service to other application users will have less control over the policy, entities, and requests directed from the application to the policy engine.

Single Host

The difficulty of exploiting this threat is increased in the single host scenario. Given that the application is trusted, attackers attempting to create a denial of service against the policy engine and/or application will have less control over the policy, entities and requests directed to the policy engine. An attacker attempting to deny service to other users is limited primarily to the privileges of a user. Given the resource coupling between the policy engine and application, an external attacker that disrupts the policy engine also denies *application* service to other users, and may even fully crash or hang the host on which the application runs, potentially also causing loss of in-memory or in-flight data.

Threat Group 3: Policy Effects Enabling Confused-Deputy Attacks

Group 3 aggregates threats where abuse of policy features could trick the policy engine or a policy-protected application to become a "confused deputy", that is, to take authorized secondary actions that require privileges the attacker does not have.

Language Features and Maturity Evaluation

Language designers can limit the likelihood of policy evaluation inducing confused-deputy interactions by maintaining pure evaluation by default and by externalizing any required authorization checks to the Policy Information Point and Policy Retrieval Point themselves.

Both OpenFGA and Cedar (with the language extensions included in the evaluated versions) do not allow side-effecting operations, which limits the likelihood that evaluation itself will result in further actions on behalf of the policy. A mismatch in privileges for the policy's data sources within the deployment would have to occur for the policy to have access to data inappropriately.

On the other hand, Rego *does* resolve network requests and external references within a policy. This functionality could allow a compromised application maintainer, or potentially even a compromised application user, to access internal resources of the policy engine or to access other applications' resources. These effects are partially mitigated through use of a capabilities list to restrict the use of default language built-ins.

Threat Scenarios	**Mitigation Maturity Summary**	**Recommended Mitigations**
TOB-T2, TOB-T5, TOB-T6, TOB-T10, TOB-T11	Cedar: Strong Rego: Moderate OpenFGA: Strong *Overall: Strong*	• Default language purity • Disallow custom runtime operations originating from policies that would leave language engine boundaries • Disallow custom runtime network operations originating from policies • Secure code review

We break down the above summary by policy language and compare features that weaken as well as improve the language with respect to Threat Group 3 below.

Cedar

Threat	**Contributory Features**	**Mitigating Features**	**Maturity Evaluation**
TOB-T2	N/A	• Pure policy evaluation by default	*Strong*: Cedar does not allow policy writers to add additional information to logs directly from policies written in the policy language at runtime, preventing data leakage beyond the logging

			performed by the deployment and policy engine itself.
TOB-T5	N/A	• Templating reduces duplication and distribution of policies	*Strong*: External resources cannot be referenced and resolved directly within a policy or from the current operations available in Cedar. External information enters the language runtime only through the entity store, request, policy, and context.
TOB-T6	N/A	• Pure policy evaluation by default	*Strong*: The Cedar policy language does not have the capability to perform side effects that would directly read or resolve internal resources.
TOB-T10	N/A	• Pure policy evaluation by default	*Strong*: The available features in the Cedar language do not allow for side effects that could directly impact the contextual information of another application.
TOB-T11	• Errors can leak sensitive entity identifiers and values	• Pure policy evaluation by default	*Strong*: The available features in the Cedar language do not allow for reading information not directly provided in the request, entity store, or context. Errors can provide further

			information from a policy than a policy decision, but error information should be restricted to the offending policy or entity.

Rego

Threat	Contributory Features	Mitigating Features	Maturity Evaluation
TOB-T2	• `Trace` allows arbitrary values to appear in a query explanation • `Print` allows arbitrary values to be printed during evaluation	• Capabilities restrict built-ins to an allowlist	*Moderate*: Rego provides policy writers with mechanisms for annotating the log with arbitrary data and printing information from within a policy at runtime to the evaluator's output stream. This functionality can be limited through use of capabilities by the authorization provider(s).
TOB-T5	• Import-based policy construction includes "base" documents that can have arbitrary contents and can be async pushed to the policy store or pulled via bundle configurations • Custom language-level extensions to Rego are supported	N/A	*Moderate*: The Rego language and configuration provides multiple features for including and resolving external resources at runtime through bundles or HTTP requests. External resources can be imported directly into a policy.

TOB-T6	• Import-based policy construction includes “base” documents that can have arbitrary contents and can be async pushed or pulled into OPA via Rego’s bundle configuration • Custom language-level extensions to Rego are allowed	• Capabilities restrict built-ins to an allowlist	*Moderate*: Current Rego built-ins can reference and resolve external resources directly from the policy language, but this weakness may be mitigated by a strong, built-in allowlist. Further complications exist if a malicious actor can gain control over built-in allowlisting.
TOB-T10	• Rego built-ins currently allow for side effects including HTTP requests and DNS lookup • Rego extensions could allow for modification of system data	• Policies can optionally be contained within a wasm sandbox • Capabilities restrict built-ins to an allowlist	*Moderate*: Built-ins in Rego allow for performing network requests that could impact the contextual information of another application.
TOB-T11	• Rego built-ins currently allow for side effects including HTTP requests and DNS lookup • Rego extensions could allow for reading of system data • Non-Boolean output requests	• Capabilities restrict built-ins to an allowlist	*Moderate*: The same built-ins and language properties from TOB-T10 could allow for reading contextual information.

	allows for rich data extraction (i.e., arbitrary document values)		

OpenFGA Modeling Language

Threat	Contributory Features	Mitigating Features	Maturity Evaluation
TOB-T2	N/A	• Pure model evaluation by default	*Strong*: OpenFGA models do not allow policy writers to add additional information to logs directly from policies written in the modeling language. Logs are restricted to the logging performed directly by the deployment itself.
TOB-T5	N/A	• Models are immutable	*Strong*: OpenFGA models are provided as a single complete module and can consume only further resources that are added to the tuple store.
TOB-T6	N/A	• Pure model evaluation by default	*Strong*: OpenFGA does not allow side-affecting operations within a model. The model expresses only relation definitions, so resources would need to be added to the store in order to be referenced by a model.
TOB-T10	N/A	• Pure model	*Strong*: Models can affect only recorded indirect

		evaluation by default	relations, which cannot produce a side effect that impacts the contextual information of another application, assuming the applications in question rely on separate stores.
TOB-T11	• Non-Boolean output requests allows for richer data extraction ie. object lists, relation lists	• Pure model evaluation by default	*Strong*: Models cannot perform side-affecting operations that could allow direct access to another application's contextual information. If an attacker does, through other means, manage to query the relations of another application, the expressive queries permitted by the OpenFGA interface could allow for significant information leakage.

Deployment Scenario Applicability
We briefly map this threat group to each of the three deployment scenarios under which we considered the three policy language engines.

Request spoofing (e.g., server-side request forgery and similar) or even information disclosure through responses to crafted requests (e.g., local file inclusion, taking advantage of inappropriately verbose error messages, and similar) could result in an external attacker or a malicious user gaining inappropriate access *through* the policy language engine, the policy validator, or another policy language system component to further services or data.

Multi-Tenant

Since multiple disjointed, mutually distrustful applications are tenants of the policy language system, such further services and/or data accessed could pertain to applications and users distinct from those which the attacker initially exploited.

An internal attacker or a compromised authorization provider could also maliciously leverage the data required for the client authorization and allowlisting present in the policy language engine and/or policy validator to connect *out* to, or to attack, any other applications backed by the policy language system.

Single Application

Since multiple mutually distrustful users may leverage the single application backed by the policy language system, any further services and/or data the attacker gains access to could pertain to multiple users. Since the application is trusted, the impact and capabilities of an attacker in this scenario are limited to features accessible through user application requests.

Single Host

Since multiple mutually distrustful users may leverage the application that embeds the policy language engine, such further services and/or data could pertain to multiple users. Additionally, if the application runs on a multi-tenant *host*, the attacker could potentially also maliciously connect to or obtain data from applications and users that run on the host outside the scope of the policy language system. Since the application is trusted, the impact and capabilities of an attacker in this scenario are limited to features accessible through user application requests.

Threat Group 4: Runtime Safety

Threat Group 4 includes threats to the policy engine that could compromise the integrity of its runtime. Such weaknesses include memory corruption and insecure deserialization. An attacker might exploit any weaknesses in this threat group using appropriately crafted malicious inputs to gain control of the policy evaluation engine.

Language Features and Maturity Evaluation

In general, the evaluated policy languages take significant steps to mitigate these threats. All compared policy languages were implemented in memory-safe programming languages. Cedar also has fuzz tests for critical attack surfaces, and Rego policies can be compiled to web assembly (wasm) in order to be run inside a sandboxed environment. Code review, linting, and static analysis could increase security posture of a given implementation with respect to these threats.

Threat Scenarios	**Mitigation Maturity Summary**	**Recommended Mitigations**
TOB-T8, TOB-T9	Cedar: Strong Rego: Strong OpenFGA: Strong	● Use of memory safe languages ● Fuzzing ● Secure code review

	Overall: Strong	

We break down the above summary by policy language and compare features that weaken as well as improve the language with respect to Threat Group 4 below.

Cedar

Threat	Contributory Features	Mitigating Features	Maturity Evaluation
TOB-T8	• Unsafe dependencies could allow for memory corruption	• Fuzzing and property based testing of interfaces • Uses safe Rust in the implementation	*Strong*: Cedar uses safe Rust, and has fuzzing results for critical surfaces including the evaluator, validator, and formatter.
TOB-T9	• All features from TOB-T8	• All features from TOB-T8	*Strong*: Same mitigating and contributing factors for TOB-T8.

Rego

Threat	Contributory Features	Mitigating Features	Maturity Evaluation
TOB-T8	• WASM compiler includes memory unsafe implementations of built-ins, however, impact will be limited to the affected WASM module's capabilities. • User registered Go built-ins could use unsafe features of Go such as unsafe	• Policies can optionally be contained within a wasm sandbox • Uses a memory safe language for implementation	*Strong*: Rego's top-down evaluator is written in Go. As of the evaluated Rego version, there are no direct usages of unsafe pointer operations. The WASM Rego implementation does include C and C++ implementations of built-ins, but WASM sandboxing restricts the effect of compromise to the WASM module,

	pointer operations • Unsafe Go operations in dependencies could allow memory corruption		depending on the WASM runtime.
TOB-T9	• All features from TOB-T8	• All features from TOB-T8	*Strong*: Same mitigating and contributing factors for TOB-T8.

OpenFGA Modeling Language

Threat	**Contributory Features**	**Mitigating Features**	**Maturity Evaluation**
TOB-T8	• Unsafe Go operations dependencies could allow memory corruption	• Uses a memory safe language for the implementation	*Strong*: OpenFGA is written in Go and has no direct usages of unsafe pointer operations in the evaluated version.
TOB-T9	• All features from TOB-T8	• All features from TOB-T8	*Strong*: Same mitigating and contributing factors for TOB-T8.

Deployment Scenario Applicability

We briefly map Threat Group 4 to each of the three deployment scenarios under which we considered the three policy language engines.

If an exploit relating to this threat group results in access to the full contents of the main (exploited) engine or validator process, for all of the below deployment scenarios, the attacker could, in addition to specifics detailed below, gain indiscriminate access to process memory or the runtime. Such access could then, for example, result in loss of control over sensitive values like keys, certificates, and secrets used to run and deploy the exploited software.

Multi-Tenant

Particularly if the policy execution engine includes a cache or local embedded data store of policies, or caches recently made decisions and/or recently received requests, an attacker who can cross the trust boundary from an application to the policy language engine with a crafted request (containing e.g., a deserialization or memory corruption exploit) could potentially in the worst case obtain not only the recent history of policy decisions across multiple applications, but potentially also obtain data from in-flight decisions and identity/entity data relating to other users of the same application, and users of other applications.

An attacker who can submit a crafted request to the *validator,* on the other hand, could in the worst case (particularly if the validator caches recently validated policies or can directly update or fetch policies from any policy repository) gain access to all policies and therefore come to better understand the resources and privileges within the system, and/or even modify any access control policies.

Single Application

An attacker who can exploit a language engine that (as above) caches or stores data locally may gain access to the data of other users of the application backed by the engine, to in-flight decision requests, and/or potentially also to identity/entity store data (depending on how and where the identity provider store runs). If, alternatively, the attacker is able to exploit the validator, in the worst case (particularly if the validator can update policies within or fetch policies from the policy repository) the attacker may gain access to create, update, and delete any policies, as under the multi-tenant model, though such policies would pertain only to the resources owned by users of the single application. Given that the policy engine trusts the application, exploitation difficulty may increase by limiting an attacker's capabilities to those of an application user.

Single Host

While, as mentioned above, many types of exploits within this threat group could result in low-level runtime or process memory access, the embedded nature of the policy engine within the application reduces the need for one or more network and/or web hops for the attacker to bridge the gap from the application to the policy engine (or the validator, if it is also run in an embedded fashion) in order to exploit it. Given that the policy engine trusts the application, exploitation difficulty may increase by limiting an attacker's capabilities to those of an application user.

Threat Group 5: Data Normalization

Group 5 consists of authorization bypasses resulting from insufficient or inconsistent data normalization between parsers. Such attack vectors are present when several different parsers exploitably handle data in subtly different fashions, when all parsers should consider the data equivalent. This type of weakness is commonly termed a *parser differential*.

These types of vulnerabilities can also occur in policy language systems when supporting policy information is parsed in an unexpected way and results in incorrect authorization decisions. For instance, an URL denylist might be circumvented by passing a URL that is equivalent to a denylisted target, but is not parsed to an equivalent form by the policy language or by some other interface providing information to the policy language.

Language Features and Maturity Evaluation

Whenever possible, particularly for common but complex data types like IP addresses, URLs, or timestamps, it is better to use commonly accepted/popular parsing and validation implementations, rather than deviating from commonly accepted practice or using partial implementations.

Current mitigations within policy languages include providing built-ins for commonly compared types such as IP addresses (provided by CEL binary bindings and extensions in OpenFGA, maintainer-created extension types in Cedar, and built-ins in Rego). If an application maintainer or authorization provider needs a type not supported by their policy language, they must normalize this data themselves. This can occur within the current boundaries of the language engine through producing appropriate records and policy comparisons in Cedar, producing appropriately normalized documents and comparisons in Rego, or appropriate records and comparisons in CEL for OpenFGA conditions. Alternatively, policy writers can also opt to extend the languages themselves with new data types and built-in comparisons. This secondary option would be complex, and results in ad-hoc language semantics for such use cases.

Languages can provide better support for developing comparable types within the policy language by allowing policy writers to define reusable comparison operators for normalized types (e.g., allowing the definition of pure functions for comparing records). Rego and OpenFGA's CEL extension allow for defining reusable functions or rules that can produce these comparisons, whereas Cedar lacks this capability directly. Structured support libraries in all the examined languages could increase the uniformity with which data normalization and comparisons are implemented.

Threat Scenarios	**Mitigation Maturity Summary**	**Recommended Mitigations**
TOB-T7	Cedar: Moderate Rego: Moderate OpenFGA: Weak *Overall: Moderate*	● Language type extensions for commonly used data types ● Documented best practices for comparing common types of data ● Use standard functionality such as that of Golang or Rust, rather than hand-rolling data normalization or common structured

		data parsing • Lints • Parser code reviews • Negative tests, error tests • Parser fuzzing

We break down the above summary by policy language and compare features that weaken as well as improve the language with respect to this threat group below.

Cedar

Threat	Contributory Features	Mitigating Features	Maturity Evaluation
TOB-T7	• Language maintainer-created extension types currently include an IP address parsing module that extends Rust standard library IP parsing with a distinct Cedar maintained CIDR parsing implementation	• Policy validator appropriately filters on and can uniquely (correctly) interpret a small subset of (but not all) unclear, invisible, or substantially similar Unicode characters in policies • IP parsing is delegated to Rust net libraries aside from changes to prefix parsing	*Moderate*: Cedar provides parsing utilities for IPv4 and IPv6 addresses and performs Unicode security checks within the validator. Various further Unicode features and normalization still may be required for entities and entity values themselves (e.g., URL parsing and normalization).

Rego

Threat	Contributory Features	Mitigating Features	Maturity Evaluation
TOB-T7	• `net`, `cidr`, and other Rego builtins are partially hand-rolled instead of closely based on the standard Golang functionality	• Use of some Golang standard primitives for IP address and CIDR block parsing	*Moderate*: Rego has a large library of parsing utilities available within the set of available built-ins. These parsing utilities generally expose well-known Go libraries to the Rego environment. For some parsing routines, the built-ins contain hand-rolled parsing code that does not have security tests.

OpenFGA Modeling Language

Threat	Contributory Features	Mitigating Features	Maturity Evaluation
TOB-T7	• Identifiers are uninterpreted strings, allowing entirely non-conformant IP addresses when used in a relation's context	• CEL condition parsing uses Golang standard primitives for handling common but hard to parse structured data such as IP addresses and timestamps	*Weak*: OpenFGA objects and users are strings resulting in an overly flexible data model on tuples. To describe relations correctly, the user must normalize all involved data into strings (e.g., representing IP addresses as strings) before inserting or checking tuples. Conditions can allow CEL expressions that operate over object context, but objects themselves are restricted to strings. The IP address and timestamp

			types that CEL provides do use Go standard library types for parsing and representation. Conditions are a new feature without support in some SDKs and OpenFGA tooling.

Deployment Scenario Applicability

We briefly map Threat Group 5 to each of the three deployment scenarios under which we considered the three policy language engines.

Multi-Tenant

While relevant to all deployment scenarios, as with Threat Group 3, the multi-tenant deployment scenario is the most broadly affected by this threat group. Effects could include, but are not limited to, an attacker exploiting unclear or insufficient Unicode (or other character or digit-encoding) handling in order to gain access to resources belonging to other users, or access to read and update the policies of other users.

The attacker could even potentially gain unauthorized and unexpected access to other *applications* (tenants) allowlisted to access the policy engine if the engine is deployed in a fashion that uses the engine's own policies for gating decision-request engine access.

Single Application

An attacker exploiting unclear or insufficient Unicode or other character or digit-encoding handling in the policy engine or the validator could gain access to resources belonging to other users or users of other applications, or access to read and update the policies of other users.

Single Host

This scenario is not currently understood to be differently exploitable from the single-application scenario for this threat group.

Threat Group 6: Policy Version and Dependency Management

Threat Group 6 includes weaknesses in policy versioning and policy dependency management. Languages themselves often have little control over authorization checks when uploading a policy, but certain language features can increase or decrease the difficulty of implementing secure policy management and versioning.

Specifically, the capability for a language to directly reference and resolve external policies increases the likelihood that an attacker can compromise a policy's dependencies and affect authorization requests. This likelihood is increased when a policy's rules are not independent and isolated.

Language Features and Maturity Evaluation

Mitigations for these types of issues include the capability to pin policy evaluation to a given version of the policy, `forbid` rules overriding any new `permit` rules, and the language allowing the use of only internal, well-defined dependencies.

For instance, Rego's incomplete rules can be extended by a dependency without directly referencing it, through defining another file within that same package that extends the rule. Similarly, when leveraging Rego's capability to extend the evaluator with additional Go built-ins, bindings to new built-ins replace old definitions without error, potentially allowing an attacker that controls Go extensions to alter how previously-defined policies will be evaluated.

Threat Scenarios	Mitigation Maturity Summary	Recommended Mitigations
TOB-T12, TOB-T13, TOB-T14, TOB-T15	Cedar: Moderate Rego: Weak OpenFGA: Moderate *Overall: Moderate*	• Enforce immutable policy versions • Allow applications to specify the required policy version • Disallow external references entirely, or enable only allowlisted references to externally provided policies • Ensure appropriate cryptographic protection of provided policies

We break down the above summary by policy language and compare features that weaken as well as improve the language with respect to Threat Group 6 below.

Cedar

Threat	Contributory Features	Mitigating Features	Maturity Evaluation
TOB-T12	N/A	• Affecting policies tracking	*Weak*: The language in and of itself does not imply any strong

			properties with respect to policy modification. Policy evaluation does track the impacting policies, providing a reasonable triaging capability to identify the offending polic(ies) that permitted a malicious or inappropriate request.
TOB-T13	N/A	• Affecting policies tracking	*Moderate*: Policy addition could allow an attacker to achieve unintended access. Policy evaluation does track the impacting policies, providing a reasonable triaging capability to identify offending policies that permitted a malicious or inappropriate request. `forbid` always overriding `permit` prevents new `permit` policies from breaking `forbid` policies. However, if there is not an applicable `forbid` policy, adding a `permit` policy should be sufficient to achieve inappropriate resource access.
TOB-T14	N/A	• Affecting policies tracking • `forbid` overrides `permit` • `forbid` by default	*Moderate*: Policy dependencies in Cedar behave like adding a new policy (either a `forbid` or `permit` policy). The same mitigating features

			apply.
TOB-T15	N/A	• Statically declared language operations without the option for runtime extension	*Strong*: Extension types do provide a way for language maintainers to easily extend Cedar with new semantic types and built-in functions. However, in their current implementation, these extensions are statically compiled in the artifact, making malicious replacement more difficult.

Rego

Threat	Contributory Features	Mitigating Features	Maturity Evaluation
TOB-T12	• Module imports allow for indirect references to policies • Incremental definitions extend rule results	N/A	*Weak*: Rego allows for importing policies from the data store, which can be modified directly or through use of bundles. An ineffectual policy can easily be constructed and can replace desired authorization behavior if the attacker has the privileges to control or modify a policy source.
TOB-T13	• Module imports allow for indirect references to policies • Incremental definitions extend	N/A	*Moderate*: Rego allows for importing policies from the data store which can be modified directly or through bundles. Added policies can impact the

	rule results		evaluation of any policy that uses incrementally extendable rules. A set or list can be extended by a new policy, or create an unexpected undefined value, potentially allowing unintended access.
TOB-T14	• Module imports allow for indirect references to policies • Incremental definitions extend rule results	N/A	*Weak*: Modifying rule dependencies rather than adding a new policy grants additional capabilities to an attacker beyond extending indirect rules. By modifying rules that a Rego policy depends on, an attacker can directly impact the behavior of the target policy.
TOB-T15	• Go built-ins allow extending Rego with arbitrary Go operations • Register builtin class can override bindings for arbitrary built-ins	• Capabilities restrict built-ins to an allowlist	*Moderate*: Rego allows for registering new built-in names within the Go SDK. These custom built-ins can replace the standard built-ins, or can be user- or attacker-defined built-ins containing arbitrary functionality.

OpenFGA Modeling Language

Threat	**Contributory Features**	**Mitigating Features**	**Maturity Evaluation**
TOB-T12	N/A	• Models are	*Moderate*[1]: Model

		immutable and can be referenced by an identifier	modification is not possible directly in OpenFGA. However, outside of the policy language itself, if an application does not pin an identifier, uploading new models to the OpenFGA. **1.** OpenFGA model immutability is partially a property of the server deployment. The OpenFGA server is tightly integrated with how users interact with the modeling language, making it unclear in this case how to separate language features from deployment features.
TOB-T13	N/A	• Models are immutable can be referenced by an identifier	*Moderate*: Model versioning and pinning identifiers can to some extent limit the impact of uploading new models maliciously, but an attacker may be able to affect components that do not refer to a model by identifier.
TOB-T14	N/A	• Models are immutable can be referenced by an identifier	*Strong*: All dependencies for a model module are specified prior to uploading, so the platform does not support changing a model's dependencies.
TOB-T15	N/A	• Uses of CEL extension functions and bindings are statically declared by OpenFGA	*Strong*: OpenFGA conditions do use CEL extension functions to build IP address types for conditions, but these functions are statically declared in the CEL

			environment initialization.

Deployment Scenario Applicability

We briefly map Threat Group 6 to each of the three deployment scenarios under which we considered the three policy language engines.

When dependency management, standard library, or modular inclusion system features of a policy language enable a naive or compromised application maintainer or authorization provider team member to include/import/compose stored policy fragments and functionality sourced from distributed locations, the likelihood that an external attacker could compromise (e.g., maliciously modify/loosen, entirely hijack) one or more of those fragments in a fashion that still validates without the application maintainers or authorization providers noticing increases.

Multi-Tenant

Distributed policy fragment source locations in the modeled multi-tenant deployment scenario could include the application policy store, the policy execution engine (e.g., if a local cache or other embedded data store exists within its boundaries), the generalized policy repository backing the policy execution engine, or any [out of scope] remote language-specific central repository of standard or potentially also user-uploaded functionality.

Under the multi-tenant model, compromise of a particular included policy module, standard library function, or policy fragment resulting in execution of unknown and potentially inadequate or outright malicious policy could lead to an attacker gaining a persistent, quiet foothold within the multi-tenant system. Such a foothold could allow them, in the worst case, potentially indiscriminate access to any and all resources (belonging to users of any tenant application) to which policies in the system apply.

Single Application

While we do combine the policy repository and the application policy store into a single repository for purposes of simplicity under this deployment strategy, distributed policy fragment source locations could still include the policy repository, the policy execution engine cache, and any (out of scope) remote language-specific central repository of standard or potentially also user-uploaded functionality.

Compromise of a particular included policy module, standard library function, or policy fragment resulting in execution of unknown and potentially inadequate or outright malicious policy could lead to an attacker gaining a persistent, quiet foothold within the single-application policy language system deployment. As above, such a foothold could potentially worst case allow the attacker indiscriminate access to any and all user-owned resources to which policies in the system apply.

Single Host

This scenario is not currently understood to be differently exploitable from the single-application scenario for this threat group.

Threat Group 7: Network and Deployment Security

Threat Group 7 captures threats related to the deployment of the policy language and communication between the policy enforcement and policy decision points, outside the scope of Threat Group 2.

These types of threats include instances where the communication channels between the Policy Decision Point and Policy Enforcement Point are not protected cryptographically, allowing a machine-in-the-middle to modify the response of the Policy Decision Point in order to produce the desired authorization effect. This category of threats, in general, cannot be directly mitigated by policy language features and therefore was not part of the language feature to threat group mapping directly, but we describe it here for the purpose of completeness.

Language Features and Maturity Evaluation

As noted in the section overview, we *did not* consider existing language features directly with respect to this threat group, since we did not find or conceive of language features that could directly mitigate deployment vulnerabilities. Such mitigations are not directly comparable at only the language level; however, potential mitigations do exist at the policy engine level that can interact with language features, as briefly summarized at a high level in the below table.

Threat Scenarios	**Mitigation Maturity Summary**	**Recommended Mitigations**
TOB-T16	Cedar: Weak Rego: Weak OpenFGA: Weak *Overall: Weak*	• Cryptographic protection of requests and responses • Enforce isolation between provider and application components

Deployment Scenario Applicability

We briefly map Threat Group 7 to each of the three deployment scenarios under which we considered the three policy language engines.

Multi-Tenant

Depending on the connection encryption the policy engine requires of its tenant applications, competing (mutually distrustful) compromised application maintainers could create decision races between requests originating from different applications by, e.g., capturing and acting on knowledge gained from responses from the policy language engine intended for others. Such a compromised application maintainer(s) or an internal attacker could also modify policy engine decision responses in flight. Either of these malicious actions could result in unexpected shared resource denial for the user of a different application or even in opportunity cost, for example when authorization or access to a limited shared resource is requested.

Single Application

Resource contention between *users* of the single application in the deployment can occur, similar to the above multi-tenant scenario.

Single Host

If the individual processes and/or intra-process components of the embedded policy language engine within the application are not sufficiently isolated, even though network connections are not required for these communications to occur, an attacker who gains access to the application’s runtime environment automatically gains privileges to intercept and modify requests to and responses from the policy engine component.

Appendix A: Grouped Feature-Threat Mappings

In this appendix, we present the same language feature to threat mappings as above, but grouped by individual threat rather than by policy language. We provide this combined version of our mapping to help the reader more easily follow our comparative evaluation across languages.

Threat to Contributory Feature Mapping

Contributory features either enable an actor to exploit a threat vector, decrease the difficulty of exploiting an attack vector, or exacerbate the severity of a given threat.

Threat	Cedar Policies	Rego Policies	OpenFGA Models
TOB-T1	• Runtime errors result in skipped `forbid` policies • Runtime errors created by arithmetic operators (ie. overflow/underflow) and missing entities are not modeled by the validator	• Flexible document model • Future keyword extensions leads to diverging language versions • Improperly handled undefined values in deny rules lead to undefined result • Negation swallows undefined errors • Non-explicit forbid/allow output • Side-effecting and non-deterministic built-ins	• Arbitrary relation queries • Non-explicit forbid/allow output • Limited to directly modeling ReBAC • CEL conditions add expressivity to write ABAC policies, but increase parsing and evaluation complexity • OpenFGA policy writers can unintentionally create CEL conditions that always error when the minimum cost estimate exceeds the maximum

			bound
TOB-T2	N/A	• `Trace` allows arbitrary values to appear in a query explanation • `Print` allows arbitrary values to be printed during evaluation	N/A
TOB-T3	• Worst-case performance of policy slicing is multiplicative in the depth of the entity store hierarchy • Set operations are, in the worst case, quadratic in the size of sets in the entity store • Performing policy principal or resource bounding in when clauses prevents policy slicing on the basis of those bounds	• HTTP built-ins have unpredictable runtime determined by the endpoint, up to a provided timeout • Net built-in `lookup_ip_addr` has an unpredictable runtime, up to the timeouts of underlying resolvers • Implicit indexing makes policy performance potentially unpredictable	• Persistent storage on writes increases the potential performance impact of tuple modifications • List object requests can incur unexpected high overhead when evaluating a complex model
TOB-T4	• All features from TOB-T3	• All features from TOB-T3	• All features from TOB-T3

		• Regex built-ins combined with control of the regex could lead to catastrophic backtracking	• Intersection relations in combination with a list objects call allow a maintainer controlling the model to create expensive calls • Regex CEL conditions combined with malicious control of the regex could lead to catastrophic backtracking • CEL macros in conditions combined with malicious macro expansion control could lead to infinite macro expansion
TOB-T5	N/A	• Import-based policy construction includes “base” documents that can have arbitrary contents and can be async pushed to the policy store or pulled via bundle configurations • Custom	N/A

		language-level extensions to Rego are supported	
TOB-T6	N/A	• Import-based policy construction includes “base” documents that can have arbitrary contents and can be async pushed or pulled into OPA via Rego’s bundle configuration • Custom language-level extensions to Rego are allowed	N/A
TOB-T7	• Language maintainer-created extension types currently include an IP address parsing module that extends Rust standard library IP parsing with Cedar maintained CIDR parsing	• `net`, `cidr`, and other Rego builtins are partially hand rolled instead of closely based on the standard Golang functionality	• Identifiers are uninterpreted strings, allowing entirely non-conformant IP addresses when used in a relation’s context
TOB-T8	• Unsafe dependencies could allow for memory corruption	• WASM compiler includes memory unsafe implementations of built-ins, however, impact	• Unsafe Go operations dependencies could allow memory corruption

		will be limited to the affected WASM module’s capabilities. • User registered Go built-ins could use unsafe features of Go such as unsafe pointer operations • Unsafe Go operations in dependencies could allow memory corruption	
TOB-T9	• All features from TOB-T8	• All features from TOB-T8	• All features from TOB-T8
TOB-T10	N/A	• Rego built-ins currently allow for side effects including HTTP requests and DNS lookup • Rego extensions could allow for modification of system data	N/A
TOB-T11	• Errors can leak sensitive entity identifiers and values	• Rego built-ins currently allow for side effects including HTTP requests and DNS lookup • Rego extensions could allow for	• Non-Boolean output requests allows for richer data extraction ie. object lists, relation lists

		reading of system data • Non-Boolean output requests allows for rich data extraction (i.e., arbitrary document values)	
TOB-T12	N/A	• Module imports allow for indirect references to policies • Incremental definitions extend rule results	N/A
TOB-T13	N/A	• Module imports allow for indirect references to policies • Incremental definitions extend rule results	N/A
TOB-T14	N/A	• Module imports allow for indirect references to policies • Incremental definitions extend rule results	N/A
TOB-T15	N/A	• Go built-ins allow extending Rego with arbitrary Go operations	N/A

		• Register builtin class can override bindings for arbitrary built-ins	

Table 2: Contributory features by threat

Threat to Mitigating Feature Mapping

Mitigating language features either fully mitigate an attack vector, increase the difficulty of exploiting the attack vector in question, or decrease the severity of the threat.

Threat	**Cedar Policies**	**Rego Policies**	**OpenFGA Models**
TOB-T1	• SMT policy analysis • Static type checking • `forbid` overrides `permit` • Deny by default • Explicitly defined `permit` or `forbid` output • Formal semantics • Differential testing of evaluator against formal model • Style guide and best practices	• Policy testing framework • Default undefined decision • JSON schema checking • Style guide and best practices	• Model testing framework • Model specifies allowable relations • Style guide and best practices
TOB-T2	• Pure policy evaluation by default	• Capabilities restrict built-ins to an allowlist	• Pure model evaluation by default
TOB-T3	• Termination proof for the Lean	• Default timeout for HTTP built-ins	• Pure model evaluation by default

	formalization • Policy slicing reduces performance impact of large policy stores • Pure policy evaluation by default	• Capabilities restrict built-ins to an allowlist • When policies are restricted to the linear language fragment, policies will run with worst case linear performance • Indexed policies transform some nested comprehensions to linear time • Policy profiling tools	• Subproblem cache may reduce duplicate work in short timeframes
TOB-T4	• All features from TOB-T3	• All features from TOB-T3 • Regex implementation uses Thompson regex matching to be O(mn) in the size of the regex and input string	• All features from TOB-T3 • Global timeout settings for requests • Regex implementation uses Thompson regex matching to be O(mn) in the size of the regex and input string • OpenFGA sets a default maximum cost limit for CEL conditions • CEL sets (tested) internal

			expression recursion and maximum recursion depth limits
TOB-T5	• Templating reduces duplication and distribution of policies	N/A	• Models are immutable
TOB-T6	• Pure policy evaluation by default	• Capabilities restrict built-ins to an allowlist	• Pure model evaluation by default
TOB-T7	• Policy validator appropriately filters on and can uniquely (correctly) interpret a small subset of (but not all) unclear, invisible, or substantially similar Unicode characters in policies • IP parsing is delegated to Rust net libraries aside from changes to prefix parsing	• Use of some Golang standard primitives for IP address and CIDR block parsing	• CEL condition parsing uses Golang standard primitives for handling common but hard to parse structured data such as IP addresses and timestamps
TOB-T8	• Fuzzing and property based testing of	• Policies can optionally be contained within	• Uses a memory safe language for the

	interfaces • Uses safe Rust in the implementation	a wasm sandbox • Uses a memory safe language for implementation	implementation
TOB-T9	• All features from TOB-T8	• All features from TOB-T8	• All features from TOB-T8
TOB-T10	• Pure policy evaluation by default	• Policies can optionally be contained within a wasm sandbox • Capabilities restrict built-ins to an allowlist	• Pure model evaluation by default
TOB-T11	• Pure policy evaluation by default	• Capabilities restrict built-ins to an allowlist	• Pure model evaluation by default
TOB-T12	• Affecting policies tracking	N/A	• Models are immutable and can be referenced by an identifier
TOB-T13	• Affecting policies tracking	N/A	• Models are immutable can be referenced by an identifier
TOB-T14	• Affecting policies tracking • `forbid` overrides `permit`	N/A	• Models are immutable can be referenced by an identifier

	• `forbid` by default		
TOB-T15	• Statically declared language operations without the option for runtime extension	• Capabilities restrict built-ins to an allowlist	• Use of CEL extension functions and bindings are statically declared by OpenFGA

Table 3: Mitigating features by threat

Threat Group Mitigation Analysis

We present maturity evaluations on a scale of Weak to Strong for each language with respect to mitigations for each evaluated threat group. To compute the overall mitigation maturity, we take the maturity average over all evaluated languages and round down to the nearest maturity level.

Group Name	Included Threats	Mitigation Maturity	Generalized Mitigations
1. Language Usage and Authorization Logic Errors	TOB-T1	Cedar: Moderate Rego: Weak OpenFGA: Moderate *Overall: Moderate*	• Analysis tools • Type checking • Testing frameworks • Semantics specification • Default deny behavior
2. Denial of Service or Unexpected Performance Effects	TOB-T3, TOB-T4	Cedar: Moderate Rego: Weak OpenFGA: Weak *Overall: Weak*	• Documented worst case performance for language operations • Avoid non-deterministic language operations • Simplify evaluation in-order to provide predictable performance • Guarantee termination
3. Policy Effects Enabling Confused Deputy	TOB-T2, TOB-T5, TOB-T6, TOB-T10, TOB-T11	Cedar: Strong Rego: Moderate OpenFGA: Strong *Overall: Strong*	• Default language purity • Disallow custom runtime operations originating from policies that would leave language engine boundaries • Disallow custom runtime network operations originating from policies

			• Secure code review
4. Runtime Safety	TOB-T8, TOB-T9	Cedar: Strong Rego: Strong OpenFGA: Strong *Overall: Strong*	• Memory safe languages • Fuzzing • Secure code review
5. Data Normalization	TOB-T7	Cedar: Moderate Rego: Moderate OpenFGA: Weak *Overall: Moderate*	• Language type extensions for commonly used data types • Documented best practices for comparing common types of data • Use standard functionality such as that of Golang or Rust over hand rolling data normalization or common structured data type parsing • Lints • Parser code reviews • Negative tests, error tests • Parser fuzzing
6. Policy Version and Dependency Management	TOB-T12, TOB-T13, TOB-T14, TOB-T15	Cedar: Moderate Rego: Weak OpenFGA: Moderate *Overall: Moderate*	• Enforce immutable policy versions • Allow applications to specify the required policy version • Disallow external references entirely, or enable only allowlisted references to externally

			provided policies • Ensure appropriate cryptographic protection of provided policies
7. Network and Deployment Security	TOB-T16	Cedar: Weak Rego: Weak OpenFGA: Weak *Overall: Weak*	• Cryptographic protection of requests and responses • Enforce isolation requirements between provider and application components

Table 5: Threat groups

Side-by-Side Maturity Ratings

Based on the contributory features and mitigating features for each language, we evaluate the maturity of mitigation for each identified threat on a scale from Weak, to Moderate, to Strong. The presence of significant contributory features without sufficient mitigating features results in a Weak threat mitigation maturity rating. These maturity ratings solely relate to language features and, as discussed earlier under Scope, do not take into account mitigations or contributory features that are implemented within deployments such as OPA, the OpenFGA server, or Amazon Verified Permissions.

Threat	Cedar Policies	Rego Policies	OpenFGA Models
TOB-T1	*Moderate*: The language provides a strong formal backing for its default `forbid` behavior, and powerful analysis tools exist for users to confirm policy and language properties. Differential testing provides evidence of implementation equivalence with the semantic model. Cedar does not provide a dynamic unit testing framework for policy writers. Runtime errors can potentially result in a policy decision of `permit` that is annotated by errors.	*Weak*: The language provides schema checking, but propagates undefined results through intersecting rules, which could cause undefined-by-default policy evaluation behavior. The testing framework allows policy writers to unit test their policies. A combination of features that allow for ignoring runtime errors (the `not` operator), weak document schema enforcement, and the ability to create complex, custom non-Boolean output formats increases the possibility of writing unintentionally misconstructed policies.	*Moderate*: OpenFGA has by-default limited expressive power that presents a simple authorization model through relations. Relations are testable and fit an analyzable relational model. Possible relations can be visualized and analyzed by the policy writer. The limited expressive power of OpenFGA may require policy writers to perform further checks outside of OpenFGA or to extend the model with custom logic that may not be tested sufficiently and is subject to errors. The lack of negative tests or a defined formal semantic model increases the risk of policy misevaluation, particularly when introducing caching.

TOB-T2	*Strong*: Cedar does not allow policy writers to add additional information to logs directly from policies written in the policy language at runtime, preventing data leakage beyond the logging performed by the deployment and policy engine itself.	*Moderate*: Rego provides policy writers with mechanisms for annotating the log with arbitrary data and printing information from within a policy at runtime to the evaluator's output stream. This functionality can be limited through use of capabilities by the authorization provider(s).	*Strong*: OpenFGA models do not allow policy writers to add additional information to logs directly from policies written in the modeling language. Logs are restricted to the logging performed directly by the deployment itself.
TOB-T3	*Strong*: Cedar policies are in general efficient to evaluate and lack side effects that could run for an unbounded timeframe. An attacker with a high degree of control of the entity hierarchy can induce quadratic work during policy slicing (due to quadratic policy keys) or policy evaluation (if the policy uses set operators).	*Moderate*: The linear time language fragment of Rego and capability restrictions can to some extent help enforce strong discipline in preventing user-generated queries from causing expensive computation. However, the language's built-ins can have unpredictable runtime characteristics that may be attacker controlled.	*Moderate*: Models and relations lack side effects and should be guaranteed to terminate on requests. However, evaluation can have unpredictable performance, especially when the model under evaluation contains intersection and exclusion operators.
TOB-T4	*Moderate*: The threats of application-provided policies and entity information at runtime both contribute to the attack surface of Cedar. A malicious or naive application maintainer could create large	*Weak*: The threats of application-provided policies and documents at runtime both contribute to the attack surface of Rego. An attacker can leverage Rego's network request related built-in	*Weak*: The threat of application-provided models contributes to the attack surface of OpenFGA. Current versions of OpenFGA allow for easily constructing list requests that exhaust the global

	policy stores with many template instantiations and a large entity hierarchy to produce expensive requests.	functionality to drastically increase policy evaluation time and resource consumption.	timeout for requests by using exclusion operators.
TOB-T5	*Strong*: External resources cannot be referenced and resolved directly within a policy or from the current operations available in Cedar. External information enters the language runtime only through the entity store, request, policy and context.	*Moderate*: The Rego language and configuration provides multiple features for including and resolving external resources at runtime through bundles or HTTP requests. External resources can be imported directly into a policy.	*Strong*: OpenFGA models are provided as a single complete module and can consume only further resources that are added to the tuple store.
TOB-T6	*Strong*: The Cedar policy language does not have the capability to perform side effects that would read or resolve internal resources directly.	*Moderate*: Current Rego built-ins can reference and resolve external resources directly from the policy language, but this weakness may be mitigated by a strong built-in allowlist. Further complications exist if a malicious actor can gain control over built-in allowlisting.	*Strong*: OpenFGA does not allow side-affecting operations within a model. The model expresses only relation definitions, so resources would need to be added to the store in order to be referenced by a model.
TOB-T7	*Moderate*: Cedar provides parsing utilities for IPv4 and IPv6 addresses and performs Unicode security checks within the validator. Various	*Moderate*: Rego has a large library of parsing utilities available within the set of available built-ins. These parsing utilities generally expose well-known Go	*Weak*: OpenFGA objects and users are strings forcing an overly flexible data model on tuples. To describe relations correctly, the user must normalize all involved

	further Unicode features and normalization still may be required for entities and entity values themselves (e.g., URL parsing and normalization).	libraries to the Rego environment. For some parsing routines, the built-ins contain hand-rolled parsing code that does not have security tests.	data into strings (e.g., representing IP addresses as strings) before inserting or checking tuples. Conditions can allow CEL expressions that operate over object context, but objects themselves are restricted to strings. The IP address and timestamp types that CEL provides do use Go standard library types for parsing and representation.
TOB-T8	*Strong*: Cedar uses safe Rust, and has fuzzing results for critical surfaces, including the evaluator, validator, and formatter.	*Strong*: Rego's top-down evaluator is written in Go. There are no direct usages of unsafe pointer operations in the evaluated Rego version. The WASM Rego implementation does include C and C++ implementations of built-ins, but WASM sandboxing restricts the effect of compromise to the WASM module, depending on the WASM runtime.	*Strong*: OpenFGA is written in Go and has no direct usages of unsafe pointer operations in the evaluated version.
TOB-T9	*Strong*: Same mitigating and contributing factors as TOB-T8.	*Strong*: Same mitigating and contributing factors as TOB-T8.	*Strong*: Same mitigating and contributing factors as TOB-T8.

TOB-T10	*Strong*: The available features in the Cedar language do not allow for side effects that could directly impact the contextual information of another application.	*Moderate*: Built-ins in Rego allow for performing network requests that could impact the contextual information of another application.	*Strong*: Models can only affect recorded indirect relations that cannot produce a side effect impacting the contextual information of another application, assuming the applications in question rely on separate stores.
TOB-T11	*Strong*: The available features in the Cedar language do not allow for reading information not directly provided in the request, entity store, or context. Errors can provide further information from a policy than a policy decision, but error information should be restricted to the offending policy or entity.	*Moderate*: The same built-ins and language properties from TOB-T10 could allow for reading contextual information.	*Strong*: Models cannot perform side-affecting operations that could allow direct access to another application's contextual information. If an attacker does, through other means, manage to query the relations of another application, the expressive queries permitted by the OpenFGA interface could allow for significant information leakage.
TOB-T12	*Weak*: The language in and of itself does not imply any strong properties with respect to policy modification. Policy evaluation does track the impacting policies, providing a reasonable triaging capability to identify the offending polic(ies) that permitted a malicious or inappropriate request.	*Weak*: Rego allows for importing policies from the data store, which can be modified directly or through use of bundles. An ineffectual policy can easily be constructed and can replace desired authorization behavior if the attacker has the privileges to control or modify a policy source.	***Moderate*[1]**: Model modification is not possible directly in OpenFGA. However, outside of the policy language itself, if an application does not pin an identifier, uploading new models to the OpenFGA. **1.** OpenFGA model immutability is partially a property of the server deployment. The OpenFGA server is tightly integrated with how users interact with the modeling language, making it unclear in this case how to separate language features from deployment features.

TOB-T13	*Moderate*: Policy addition could allow an attacker to achieve unintended access. Policy evaluation does track the impacting policies, providing a reasonable triaging capability to identify offending policies that permitted a malicious or inappropriate request. `forbid` always overriding `permit` prevents new `permit` policies from breaking `forbid` policies. However, if there is not an applicable `forbid` policy, adding a `permit` policy should be sufficient to achieve inappropriate resource access.	*Moderate*: Rego allows for importing policies from the data store that can be modified directly or through bundles. Added policies can impact the evaluation of any policy that uses incrementally extendable rules. A set or list can be extended by a new policy, or create an unexpected undefined value, potentially allowing unintended access.	*Moderate*: Model versioning and pinning identifiers can to some extent limit the impact of uploading new models maliciously, but an attacker may be able to affect components that do not refer to a model by identifier.
TOB-T14	*Moderate*: Policy dependencies in Cedar behave like adding a new policy (either adding a `forbid` or `permit` policy). The same mitigating features apply.	*Weak*: Modifying rule dependencies rather than adding a new policy grants additional capabilities to an attacker beyond extending indirect rules. By modifying rules that a Rego policy depends on, an attacker can directly impact the behavior of the target policy.	*Strong*: All dependencies for a model module are specified prior to uploading, so changing a model's dependencies is not supported by the platform.
TOB-T15	*Strong*: Extension types do provide a way for	*Moderate*: Rego allows for registering new	*Strong*: OpenFGA conditions do use CEL

	language maintainers to easily extend Cedar with new semantic types and built-in functions. However, in their current implementation, these extensions are statically compiled in the artifact, making malicious replacement more difficult.	built-in names within the Go SDK. These custom built-ins can replace the standard built-ins, or can be user or attacker-defined built-ins containing arbitrary functionality.	extension functions to build IP address types for conditions, but these functions are statically declared in the CEL environment initialization.

Table 4: Threat mitigation maturity table

Appendix B: Maturity Rating Criteria

Rating Criteria	
Rating	**Description**
Strong	The language feature has no security issues and its security exceeds industry standards.
Moderate	The language feature has several security issues or an impactful issue that may expose system end users to some degree of risk, albeit not to a severe degree. Remediation in this area is recommended.
Weak	The language feature has several significant security issues that are likely to expose system end users to a substantial amount of risk. Remediation in this area should be prioritized.