

Original Article

Threat-Model-Driven Zero-Trust Authorization for Microservice APIs in Kubernetes-Based Cloud-Native Applications

Rajesh Balaji

Independent Researcher, Seattle, WA, USA,
Anna University, Chennai, India.

Received Date: 10 December 2024

Revised Date: 20 December 2024

Accepted Date: 30 December 2024

Abstract: Authorization failures in micro service APIs often arise from ordinary requests whose identity, object, or service context is not checked consistently. We investigate whether a structured threat model can be converted into executable authorization tests, rather than remaining a design document. The study uses AegisCommerce, a modeled seven-service retail API, and combines STRIDE and the OWASP API Security Top 10 to identify misuse cases. We then map those cases to Open Policy Agent (OPA) rules and configuration checks, comparing a baseline policy with a threat-informed alternative. The experiment evaluates actual OPA decisions over a fixed request universe; it does not deploy the complete microservice application on Kubernetes.

Of 820 enumerated requests, 540 were over-permitted by the baseline and 28 by the hardened policy, a 94.8% reduction. All 73 intended requests remained permitted in the modeled test set. Automated checks passed for 19 of 22 catalogued threats, versus five at baseline. None of the specified attacker starting states reached the five modeled goals after hardening, although 28 residual permissions remain when an escalated role claim is supplied. Separately measured OPA and ES256 component costs were composed into illustrative added-overhead estimates of 0.58–2.32 ms for an order read and 1.37–4.86 ms for checkout; these are not end-to-end application measurements. The results demonstrate a repeatable way to test authorization against an explicit threat model while exposing the limits of role-claim trust and model coverage.

Keywords: API Security, Microservices, Zero Trust, Authorization, OWASP API Top 10, STRIDE, Open Policy Agent, OAuth Token Exchange, BOLA, Kubernetes

I. INTRODUCTION

Most serious breaches of cloud-native applications in recent years have not needed a network exploit. The attacker logged in, or borrowed a token, and then asked an API for something it should not have handed over. A customer changes an order number in a URL and reads someone else's order. A mobile client calls an admin endpoint that was never meant to be public but was never actually protected. An image-fetch feature is pointed at an internal service and used to issue a refund. These are application problems. Firewalls, network segmentation and even a well-configured Kubernetes cluster do little about them, because every request travels along a path the application is supposed to use.

The OWASP API Security Top 10 [6] puts broken object-level authorization first and broken function-level authorization fifth, and both are about the same thing: the code that decides who may do what to which object. In a microservice application that decision is spread across many places. The gateway checks that a token is valid. Each service may or may not check what the caller is allowed to do. Internal services often trust any request that arrives from inside the cluster, so a single compromised or tricked service can act on behalf of everyone. Zero Trust guidance for microservices from NIST [2], [3] says each request should be authenticated and authorized on its own merits. Turning that sentence into concrete policy for a real application is where teams struggle.

This paper does that translation for a representative application and measures the result. We model the API surface of a seven-service retail application, AegisCommerce, and apply STRIDE [5] and the OWASP API Top 10 to every endpoint and



service-to-service call. Each threat we find is mapped to a control that a standard component can enforce: token validation at the gateway, audience-scoped tokens issued by token exchange [12], workload identity for service calls [9], request-level authorization policy in Open Policy Agent [10], object-ownership checks, schema validation, rate limits and privacy-aware logging. We then build two configurations of the same application, a common baseline and the threat-model-driven design, and compare them on what each lets an attacker do, what legitimate traffic each blocks, and what the policy evaluation actually costs.

The contributions are as follows.

- A threat catalogue for the AegisCommerce API surface that combines STRIDE with the OWASP API Top 10, including service-to-service threats such as confused-deputy calls and token replay across services.
- A reproducible threat-to-control mapping in which every threat is linked to a control and to an automated check against the configuration, so coverage can be tested rather than asserted.
- An authorization analyser that enumerates, for every principal, endpoint and object, whether a request is allowed. It compares the result with the intended access matrix to count over-permissions, finds multi-step attack chains through the API, and replays the full set of legitimate requests to count wrongful denials.
- Component-level performance evaluation: benchmark real OPA policy decisions and ES256 signing and verification, then calculate illustrative request-path overhead from the measured components; do not conflate the calculation with live application response time.

The work does not introduce a new identity protocol or policy engine. Its contribution is a reproducible mapping from application-specific threats to executable policy checks, together with an explicitly bounded evaluation of authorization outcomes and component costs.

Section II reviews related work and Section III presents the threat model. Section IV describes the architecture and Section V the threat-to-control mapping. Section VI covers the experimental setup and Section VII the results. Section VIII discusses limitations and concludes.

II. BACKGROUND AND RELATED WORK

A. Zero Trust for Micro Service Applications

NIST SP 800-207 [1] defines Zero Trust as per-request access decisions based on identity and context rather than network location. Two later NIST publications apply it to applications. SP 800-204 [2] lists security strategies for microservice systems, including authentication at every hop, short-lived tokens and centralized policy. SP 800-207A [3] describes a model in which each service request carries both a workload identity and an end-user identity and is authorized against policy at the service. SP 800-204B [4] shows how attribute-based access control [15] can be enforced through a service mesh. These documents describe the target state well. They leave open how a team decides, for a specific application, which rules are needed and whether the rules it wrote are enough.

B. API Security

The OWASP API Security Top 10 [6], revised in 2023, catalogues the failures that dominate real API incidents. Broken object-level authorization (BOLA), broken authentication, broken object-property-level authorization, unrestricted resource consumption, broken function-level authorization (BFLA) and server-side request forgery (SSRF) are all on the list. The OWASP Application Security Verification Standard [17] turns these into testable requirements. Studies of microservice security, such as the challenges described by Yarygina and Bagged [24] and the literature review by Pereira-Vale et al. [26], keep returning to the same point: authorization is split between the gateway and the services, and the internal network is often treated as trusted. The 2019 Capital One incident [23], in which an SSRF flaw let an attacker obtain cloud credentials from an internal metadata service, is a well-documented example of what that trust costs.

C. Tokens and Service Identity

Oath 2.0 [7] and JSON Web Tokens [8] are the usual way to carry user identity into an API. A token issued for the whole application, which every service accepts, lets any service that receives it act as the user anywhere. Token exchange [12] addresses this by letting a service swap an incoming token for a narrower one aimed at a specific downstream audience. Sender-constrained tokens, bound to a client certificate [13] or a proof-of-possession key with Pop [14], make a stolen token much

harder to replay. For the services themselves, SPIFFE [9] gives each workload a verifiable identity that can be checked on every call.

D. Authorization Models and Policy Engines

Role-based access control handles function-level questions well ("may a customer call the refund endpoint?") but not object-level ones ("may this customer read this order?"). Attribute-based [15] and relationship-based models such as Google's Zanzibar [11] handle the latter by deciding on properties of the caller and the object together. Policy engines such as Open Policy Agent [10], often called from the service mesh through Envoy's external authorization interface or expressed as Istio authorization policies [20], let teams write these rules as code outside the service. Schema validation against an OpenAPI description [16] rejects malformed and over-posted requests before they reach business logic.

E. Privacy in Application Logs

Application logs are where personal data leaks most easily: request bodies, query strings with email addresses, bearer tokens in headers. GDPR's data-minimization and privacy-by-design principles [18] and the NIST Privacy Framework [19] both call for collecting only what is needed. For security logging, that means recording who called what on which object, with tokens hashed and personal fields redacted, rather than the full request.

F. Threat Modeling and Attack Chains

STRIDE [5] is the most widely used method for systematically asking what can go wrong at each element of a data-flow diagram. Attack graphs [25] show how individual weaknesses chain together. We apply both at the API level: nodes are principals, endpoints and data objects, and edges are requests the configuration allows.

G. Position of this Work

Existing standards and tools offer threat taxonomies, identity mechanisms and policy engines. The specific question addressed here is how a development team can connect those elements in a testable chain: from a named misuse case, to an enforceable authorization rule, to an automated regression check, and finally to an account of residual permissions. This is a workflow contribution, not a claim to have invented a new policy language or Zero Trust protocol.

III. THREAT MODEL OF THE API SURFACE

A. The Application

Aegis Commerce is a modeled seven-service retail API testbed intended for Kubernetes-based deployment; the complete seven-service application was not deployed as part of these experiments.

Table 1: Aegis Commerce API endpoints and intended access

ID	Service	Endpoint	Intended access
E1	identity	POST /login	Anyone
E2	catalog	GET /products	Anyone
E3	catalog	GET /products/{id}	Anyone
E4	catalog	POST /products/{id}/image (fetch image from URL)	Support; allow-listed hosts only
E5	cart	GET /cart	Owner (also orders, on the owner's behalf)
E6	cart	POST /cart/items	Owner
E7	orders	POST /orders	Customer
E8	orders	GET /orders/{id}	Owner or support
E9	orders	GET /orders? User={u}	Owner or support (also support service)
E10	orders	PATCH /orders/{id}/status	Support
E11	payments	GET /payments/{id}	Owner or support
E12	payments	POST /payments/charge	Internal only: orders, on the owner's behalf
E13	payments	POST /payments/{id}/refund	Support
E14	identity	GET /users/{id}	Owner (full) or support (masked fields)
E15	identity	PATCH /users/{id}	Owner; name and address only

E16	support	GET /support/search	Support
-----	---------	---------------------	---------

Its gateway represents client entry, identity represents login and profiles, catalog represents product browsing and image import, cart represents baskets, orders represents ordering, payments represents charging and refunds, and support represents the support console. The experiment evaluates the corresponding request model and executable OPA authorization policies.

The application exposes 16 API endpoints (Table I). Two service-to-service flows are legitimate: orders reads the cart and charges payments on behalf of the customer placing the order, and support reads orders and profiles on behalf of the support agent using the console. There are three kinds of user: customers, who may act only on their own carts, orders, payments and profiles; support agents, who may read customer records, change order status and issue refunds; and anonymous visitors, who may browse the catalog and log in.

B. Assets and Attacker Goals

We treat five outcomes as attacker goals: reading another customer's order, reading another customer's payment, charging another customer, issuing a refund, and reading another user's stored password hash. The trust boundaries are the internet edge at the gateway, the boundary between each pair of services, and the boundary between a service and the data it returns.

C. Attackers

We consider three starting points. An anonymous attacker has only internet access. A customer attacker has a legitimate account and a valid token, which is the most common starting point in real API incidents. A compromised service attacker controls one service's code or credentials, through a vulnerable dependency or an SSRF-style flaw, and can make calls from inside the cluster. Compromise of the identity provider's signing keys and of the Kubernetes control plane are out of scope.

D. STRIDE and OWASP API Analysis

We applied STRIDE [5] to each endpoint and each service-to-service flow, and labelled each threat with the matching OWASP API Top 10 category [6] where one exists. After merging duplicates, 22 threats remained. Table II summarizes them by category; Section V gives the full list with controls and checks.

Table 2: Threats by STRIDE Category

Category	Threats	Typical example in Aegis Commerce
Spoofing	5	A customer's token is replayed directly against an internal-only endpoint
Tampering	5	A customer adds role=support to a profile update (mass assignment)
Repudiation	2	An internal call carries no user, so nobody can tell whose action it was
Information disclosure	5	A customer changes the order ID in the URL and reads another customer's order (BOLA)
Denial of service	2	No rate limit on login, enabling credential stuffing
Elevation of privilege	3	A compromised service calls any endpoint as any user
Total	22	

E. Attack Chains

Single threats combine. In AegisCommerce, mass assignment (T5) turns a customer into a support agent, the support role unlocks image import (E4), and an unrestricted image fetch (T16) lets the attacker make requests from inside the catalog service, where internal trust (T3) does the rest. To find such chains we model the attacker as moving between states (anonymous, customer, escalated customer, SSRF from catalog, compromised service). Requests the configuration allows are the edges. Section VI explains how the edges come from the policy engine rather than from our reading of the design.

IV. ARCHITECTURE

The proposed deployment architecture would check each request at every hop for the calling workload, the represented end user and the requested object (Fig. 1). The implemented evaluation tests the OPA policy decisions and component-level token operations; it does not establish that all gateway, sidecar, workload-identity and token-exchange integrations were deployed together.

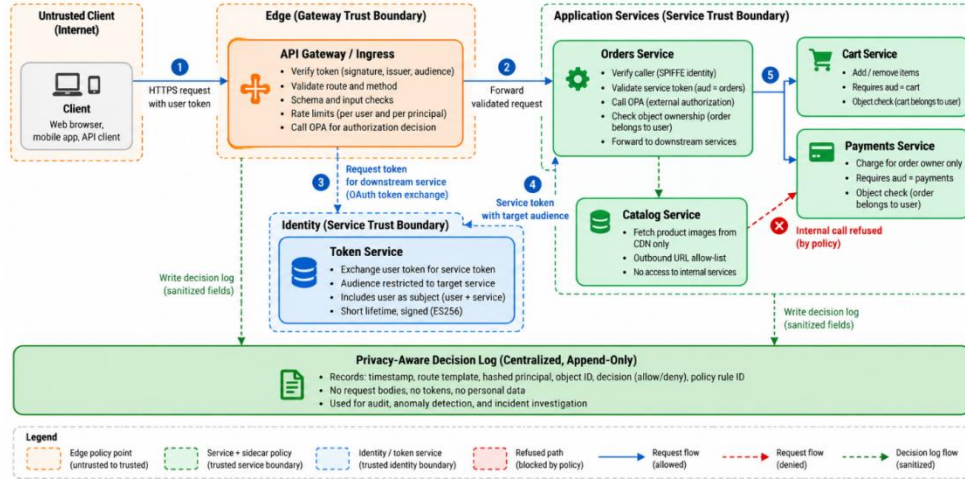


Figure 1: Checkout Request Path · Gateway, 3 Service Hops, Token Exchange, Refused SSRF Path

Figure 1. Proposed request flow and Zero-Trust enforcement boundaries in AegisCommerce. The architecture illustrates gateway validation, audience-scoped token exchange, workload identity verification, request-level authorization, restricted service-to-service communication, and privacy-aware decision logging.

A. Edge: The Gateway

In the proposed deployment, the gateway would be the only external entry point. It verifies the user's token, which must be issued for the gateway audience, and rejects any path not listed in the OpenAPI description [16], so internal endpoints such as the payment charge simply do not exist from outside. It validates request bodies against the schema, which rejects unknown or protected properties such as role. It applies a rate limit to login and a per-principal request limit, and it asks the policy engine whether this principal may call this route at all.

B. Identity between Services

Each service runs with a SPIFFE workload identity [9], and every service-to-service call is mutually authenticated. A service never forwards the user's original token. It asks the token service for a new token aimed at the specific downstream service, using OAuth token exchange [12]. The exchanged token names the user as subject and the calling service as actor, and the token service issues it only if that caller-to-target edge is in the application's call graph. A compromised catalog service therefore cannot obtain a token that payments will accept, because catalog has no legitimate reason to call payments.

C. Request-Level Authorization in Each Service

In the proposed deployment, each service's sidecar would call the policy engine through Envoy's external authorization interface [21] before the request reaches application code. The policy, about 90 lines of Rego for the whole application, checks the token audience, the caller's workload identity, the function-level rule (customer, support) and the object-level rule (the caller owns the order, payment or profile, or is support). For profile reads it also returns the list of fields the caller may see, so the service filters its response before sending it. This is object-property-level authorization in the OWASP sense.

D. Outbound Requests

The one feature that fetches a caller-supplied URL, product image import, may only reach an allow-listed content-delivery host, following the OWASP SSRF guidance [22]. Even if that check were bypassed, a request from catalog to any internal service would fail the identity and call-graph checks above. That second layer is what Fig. 1 shows as the refused path.

E. Privacy-Aware Logging

The proposed deployment would log every policy decision with the route template rather than the full URL, a hashed principal identifier, the object ID, the decision and the rule that decided it. Request bodies and bearer tokens are never logged. An investigator can still answer who did what to which object and when, but the logs cannot be used to read personal data or replay a session. This follows the data-minimisation principle in GDPR [18] and the NIST Privacy Framework [19].

V. THREAT-TO-CONTROL MAPPING

Table 3: Threat-To-Control Mapping and Automated Coverage Check

ID	STRIDE	OWASP	Threat	Control	Probes	Baseline	Hardened
T1	Spoofing	API2	Unauthenticated access to a protected endpoint	Token required at gateway and service	41	Yes	Yes
T2	Spoofing	API2	Credential stuffing against /login	Login rate limit at gateway	config	No	Yes
T3	Spoofing	-	Service impersonation inside the cluster	Workload identity + call-graph allow-list	561	No	Yes
T4	Spoofing	API2	User token used directly against an internal-only endpoint	Gateway route allow-list + audience-scoped tokens	20	No	Yes
T5	Tampering	API3	Mass assignment of the role field	Schema validation of request bodies	15	No	Yes
T6	Tampering	API1	Changing another user's profile	Object ownership check	38	Yes	Yes
T7	Tampering	API5	Customer changes order status	Function-level authorization (support role)	28	Yes	Yes
T8	Tampering	API5	Customer issues a refund	Function-level authorization (support role)	28	Yes	Yes
T9	Tampering	API6	Charging another customer's payment	On-behalf-of ownership check for charge	66	No	Yes
T10	Repudiation	-	Actions not attributable to a user	Every internal call carries an on-behalf-of user	200	No	Yes
T11	Repudiation	-	Logs lack a stable, non-reversible user reference	Hashed principal in logs	config	No	Yes
T12	Information disclosure	API1	Reading another customer's order	Object ownership check	28	No	Yes
T13	Information disclosure	API1	Reading another customer's payment	Object ownership check	14	No	Yes
T14	Information disclosure	API3	Excessive profile fields in responses	Field-level response filtering	all reads	No	Yes
T15	Information disclosure	-	Bearer tokens and bodies written to logs	Privacy-aware logging	config	No	Yes
T16	Information disclosure	API7	SSRF from image import to internal services	Fetch allow-list + internal identity	5	No	Yes
T17	Denial of service	API4	Unrestricted request rate per principal	Per-principal rate limit	config	No	Yes
T18	Denial of service	API6	Automated abuse of order creation	Out of scope: bot management	-	No	No
T19	Elevation of privilege	API5	Customer calls support search	Function-level authorization	7	Yes	Yes
T20	Elevation of privilege	-	Compromised service acts as any user	Token exchange limited to the call graph	374	No	Yes

T21	Elevation of privilege	API5	Escalated identity reaches support functions	Relies on T5 blocking escalation	9	No	No
T22	Spoofing	API2	Stolen token replayed from another client	Sender-constrained tokens (Pop or mills)	-	No	No

Table III records the trace from each threat to its proposed control and corresponding automated check. For authorization threats, the check executes representative prohibited requests against OPA; for selected configuration threats, it inspects the modeled configuration. For example, cross-customer order access is tested using non-owner principals and the applicable read operations. The coverage figure therefore means that the specified checks passed; it is not a measured fraction of real-world attacks prevented. Some checks verify configuration presence rather than deployed effectiveness.

Three threats stay open in the hardened design, and each tells us something. T18 (automated abuse of a legitimate business flow) needs bot detection, which is outside an authorization policy. T22 (replay of a stolen token from a different client) needs sender-constrained tokens [13], [14], which the design supports but the measured configuration does not enable. T21 is the most instructive. The policy trusts the role claim in the token. Schema validation stops the one escalation path we found, but if a role were ever escalated another way, for example by a bug in the identity service, the support functions would honor it. Defense in depth here means checking roles against the identity store for sensitive actions such as refunds, not only reading them from the token.

The baseline passes five checks. It enforces authentication at the gateway and has explicit role checks on a few support functions and an ownership check on profile updates. These are the controls developers tend to remember. What it misses are the controls that live between services, in response shaping and in configuration.

VI. EXPERIMENTAL SETUP

A. Two Configurations

Both configurations represent the same seven modeled services and endpoints; the experiment runs their OPA policies, not deployed services. They differ only in how requests are authenticated and authorized, and both are written as Rego policies so the same engine evaluates them.

Table 4: Configuration Differences

Aspect	Baseline	Threat-model-driven
Token accepted by services	Any application token (aid = AegisCommerce)	Token exchanged for that service (aid = service)
Gateway routing	All paths, including internal ones	OpenAPI route allow-list; internal endpoints not exposed
Service-to-service calls	Trusted if inside the cluster	Verified workload identity, call-graph allow-list, on-behalf-of user required
Object ownership	Checked on cart and profile only	Checked on every object endpoint
Request bodies	Any field accepted	Validated against schema; unknown fields rejected
Profile responses	Full record	Fields filtered by viewer (owner or support)
Image import	Any URL	Allow-listed hosts only
Rate limits	None	Login 5/min, 600 requests/min per principal
Logging	Bodies and bearer tokens	Method, route template, hashed principal, object ID, decision
Policy size	63 lines of Rego	86 lines of Rego

The baseline is an illustrative configuration reflecting several API-security failure patterns described in OWASP guidance [6] and the micro service literature [24], [26]; it is not an empirical sample of deployed organizations. Both baseline and hardened configurations are evaluated using the same OPA engine and request model. The baseline accepts application-wide audience tokens, exposes internal routes in the model, checks ownership selectively, and represents implicit trust in internal callers.

B. Request Universe and Analyzer

The analyzer generates every request the model can express: 220 external requests and 600 internal ones, 820 in total. The external requests cover four principals (anonymous, two customers and a support agent) plus an escalated customer, against all 16 endpoints and every relevant object. They include variants for protected fields in profile updates and internal URLs in image import. The internal requests cover each of six services calling every endpoint of the other services, without a user, on behalf of a customer and on behalf of a support agent. Each request is labelled with whether the intended access matrix (Table I) allows it; 73 requests are legitimate, 13 of them internal.

All 820 requests are sent to Open Policy Agent 0.68 [10] in a single evaluation per configuration, so every allow or deny decision in Section VII comes from the policy engine, not from our reading of the policy. From the decisions the analyser computes:

- Over-permissions: requests allowed but not intended, grouped by OWASP category.
- Legitimate denials: intended requests that the policy refuses.
- Sensitive field exposure: profile fields returned beyond what the viewer should see.
- Attack chains: shortest sequences of allowed requests from each attacker starting point to each goal, found by breadth-first search over attacker states.
- Threat coverage: the probe and configuration checks of Table III.

C. Overhead Measurement

We measured ES256 token signing and verification and OPA evaluation on the stated two-core Intel Xeon test environment. The token benchmark comprised 5,000 operations; OPA evaluation used five benchmark runs and 2,000 local REST requests per request type. The in-process OPA result and Python-to-OPA REST result characterize different integration paths, not guaranteed lower and upper bounds for all production systems. We combine these component timings to illustrate the cost of an order read and checkout. No real client-to-service response time, AKS networking cost, concurrent load, or remote token exchange was measured.

D. Reproducing the Results

```
python3 spec.py      # writes the data files for both policies
python3 analyze.py   # 820 requests through OPA for each configuration -> results. Son
python3 bench.py     # token and policy benchmarks -> bench. Son (about 1 minute)
python3 overhead.py  # per-request overhead from the measured parts -> overhead. Son
```

VII. RESULTS

In the enumerated request set, over-permissive decisions decreased from 540 to 28, and none of the 73 expected legitimate requests was denied. Automated threat-control checks increased from five to 19 of 22. The modeled attack-chain search found no path from its specified initial attacker states to its specified goals after hardening. The overhead estimates in Table IX are derived from measured component timings; they should not be interpreted as observed application response-time increases.

A. Over-Permission and Coverage

Table 5: What Each Configuration Allows (820 Requests Evaluated By OPA)

Measure	Baseline	Threat-model-driven
Requests allowed	613	101
Allowed but not intended (over-permissions)	540	28
... from outside, through the gateway	63	28
... from a service inside the cluster	477	0
Legitimate requests denied (of 73)	0	0
Sensitive profile fields exposed	43	0
Threats covered (of 22)	5	19

Table 6: Baseline Over-Permissions by Category

Category	Over-permissions
Service-to-service trust (confused deputy)	477
Escalated identity (customer with role=support)	29
Internal endpoint exposed at the gateway (API5/API9)	16
Broken object-level authorization (API1)	12
Mass assignment (API3)	3
Server-side request forgery (API7)	2
Broken function-level authorization (API5)	1

Nearly nine in ten baseline over-permissions come from a single assumption: that a request from inside the cluster can be trusted. This is the application-layer version of a flat network. Once any service is compromised, the attacker can call any other service and act as anyone. The external over-permissions are fewer but are the ones a customer can use with nothing more than their own login. Table VI assigns each over-permitted request to one primary failure category for counting purposes; a request may implicate additional security weaknesses.

All 28 over-permissions left in the hardened configuration belong to the escalated customer, an identity the hardened configuration never lets arise because schema validation rejects the role field. We count them anyway because they show the residual dependency of threat T21: authorization trusts the role claim in the token.

No legitimate request was denied in either configuration. This result needs a caveat. The hardened policy was written from the same access matrix the analyser checks against, so zero is partly by construction. Its real value is as a regression test, since any future policy change that breaks a legitimate request will show up immediately. Legitimate failures caused by real clients, such as a mobile app that sends an extra field the schema rejects, can only be found by testing against real traffic (Section VIII-C).

B. Attack Chains

Table 7: Shortest Attack Chain in The Baseline (Number of Requests; – = No Path; Own = The Service's Own Endpoint)

Starting point	Another customer's order	Another customer's payment	Charge another customer	Issue a refund	Another user's password hash
Anonymous	–	–	–	–	–
Customer token	1	1	1	2	2
Compromised identity	1	1	1	1	own
Compromised catalog	1	1	1	1	1
Compromised cart	1	1	1	1	1
Compromised orders	own	1	1	1	1
Compromised payments	1	own	own	own	1
Compromised support	1	1	1	1	1

For the modeled hardened policy, the search found no path from the enumerated starting states to the specified goals. In the baseline model, several goals were reachable in one or two requests, including object access across customer boundaries and role-field manipulation followed by a privileged operation. The result is conditional on the modeled state transitions and entry points. In particular, supplying an already escalated role claim reveals 28 remaining over-permissions; the path search does not establish that such a claim is impossible in an operational identity system.

The hardened result has one limit that the model makes visible. A compromised service can still act within its legitimate call graph for any user whose request it is currently handling. A compromised orders service can still charge the card of the customer whose checkout it is processing. Token exchange narrows what a compromised service can do. It does not make a compromised service harmless.

C. Measured Overhead

Table 8: Measured Enforcement Cost (Median; 99th Percentile In Brackets)

Component	Baseline	Threat-model-driven
ES256 token verification	0.172 ms (0.289)	0.172 ms (0.289)
ES256 token signing (per token exchange)	–	0.046 ms (0.093)
OPA evaluation, in process (opal bench)	0.127 ms	0.180 ms
OPA decision via REST, owner-checked read	1.01 ms (2.62)	1.05 ms (2.94)
OPA decision via REST, profile with field filter	1.05 ms (2.92)	1.13 ms (3.38)

Table 9: Added Latency Per Request Path (Median)

Request path	Hops (verify / exchange / decide)	Baseline	Threat-model-driven	Added
GET /orders/{id}	2 / 1 / 2	0.17 ms	0.75 to 2.49 ms	0.58 to 2.32 ms
POST /orders (checkout)	4 / 3 / 4	0.17 ms	1.54 to 5.03 ms	1.37 to 4.86 ms

The lower-bound calculation uses measured OPA in-process evaluation; the upper-bound calculation uses a local OPA REST round trip measured from Python. Both are composed with measured ES256 operations to estimate the additional enforcement work for illustrative request paths, not to measure live HTTP request latency. The extra OPA evaluation cost is about 0.05 ms per decision in the tested setup. Real Envoy integration, token-service network calls, database operations, caching, concurrent load and client behavior could materially change end-to-end performance. These effects require separate deployment tests.

VIII. DISCUSSION

A. What the Method Gives a Development Team

The main practical result is a repeatable authorization regression test. A team can express an intended boundary as a request that must fail—for example, one customer reading another customer’s order, or a service other than orders attempting a charge—and execute that request against the policy during continuous integration. This makes the relationship between a threat-model entry and a concrete authorization decision inspectable. It does not, by itself, prove that every application route actually invokes the policy engine.

The largest category of baseline over-permissions concerned service-to-service trust. This suggests that reviewing endpoint permissions in isolation can miss authority inherited through internal calls. The model also exposed risks involving unexpected role fields, response shaping and outbound destinations. These observations concern the tested application model; they should not be generalized into claims about how all development teams implement security controls.

B. Limitations

- Policy, not code. The analyser checks the authorization policy. If a service has a code path that bypasses the policy enforcement point, for example a debug endpoint that never calls OPA, the analysis will not see it. The method assumes every endpoint is wired to the policy, and that assumption needs its own check.
- One modelled application. AegisCommerce has seven services and sixteen endpoints. Real applications have hundreds of endpoints and evolving call graphs, where keeping the model current is itself work.
- Legitimate failures by construction. As noted in Section VII-A, zero wrongful denials partly follows from writing the policy from the same matrix. Failures caused by real client behavior need live testing.
- Overhead environment. Measurements come from a two-core virtual machine and a Python client. Token exchange is signed locally. A production deployment with Envoy, a remote token service and real network hops will differ, probably with faster policy calls and slower exchanges.

- Open threats. Token replay from another client (T22), automated business-flow abuse (T18) and the dependence on the role claim (T21) remain open in the measured configuration.

C. Validating Against Live Traffic

A subsequent integration study should deploy a working application and both configurations on an isolated test cluster, verify that every endpoint reaches its policy enforcement point, and replay representative legitimate and adversarial requests. That study should separately report policy mismatches, wrongful denials, p50/p99 application response times, token-service overhead and authorization bypass attempts. No such live measurements are reported here.

D. Future Work

Static authorization leaves a known gap: requests that the policy correctly allows can still be malicious. Examples are a customer enumerating their own data at machine speed, a support agent issuing refunds to accounts they control, or a stolen but valid session. Closing that gap needs runtime detection at the request level, with the ability to tighten authorization while an incident unfolds. That is the subject of our follow-on work.

IX. CONCLUSION

We examined whether API threat modeling can produce an executable authorization test suite for a seven-service retail application model. Across 820 enumerated OPA decisions, the hardened policy reduced over-permissions from 540 to 28 while allowing all 73 modeled legitimate requests. Threat-control checks passed for 19 of 22 catalogued threats. The attack-chain analysis found no reachable modeled goal from its specified starting states, but 28 residual permissions under an externally supplied escalated role claim remain an important limitation. Component benchmarks support illustrative added enforcement-overhead estimates, not deployed Kubernetes latency claims. These findings support threat-to-policy traceability as a useful engineering practice; broader effectiveness requires independent workloads, end-to-end enforcement checks and live-traffic validation.

Evidence and availability. The authorization counts and threat coverage reported here are results of the supplied AegisCommerce/ShopFlow OPA evaluation model. The name of the testbed does not change the evaluated artifacts. To reproduce the study, retain the original executable policy files, request generator, tool versions and random seeds alongside any renamed manuscript. No production retail data or deployed seven-service cluster is claimed. Any source-code release or persistent archive identifier must be added only after publication of the actual package.

X. REFERENCES

- [1] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero Trust Architecture," NIST Special Publication 800-207, Aug. 2020.
- [2] R. Chandramouli, "Security Strategies for Microservices-based Application Systems," NIST Special Publication 800-204, Aug. 2019.
- [3] R. Chandramouli and Z. Butcher, "A Zero Trust Architecture Model for Access Control in Cloud-Native Applications in Multi-Location Environments," NIST Special Publication 800-207A, Sep. 2023.
- [4] R. Chandramouli, Z. Butcher, and A. Chetal, "Attribute-based Access Control for Microservices-based Applications Using a Service Mesh," NIST Special Publication 800-204B, Aug. 2021.
- [5] A. Shostack, Threat Modeling: Designing for Security. Indianapolis, IN, USA: Wiley, 2014.
- [6] OWASP Foundation, "OWASP API Security Top 10 – 2023," owasp.org.
- [7] D. Hardt, Ed., "The OAuth 2.0 Authorization Framework," RFC 6749, IETF, Oct. 2012.
- [8] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, IETF, May 2015.
- [9] SPIFFE Project, "SPIFFE and SPIRE," spiffe.io.
- [10] Open Policy Agent Project, openpolicyagent.org.
- [11] R. Pang et al., "Zanzibar: Google's Consistent, Global Authorization System," in Proc. USENIX Annual Technical Conference (ATC), 2019.
- [12] M. Jones, A. Nadalin, B. Campbell, J. Bradley, and C. Mortimore, "OAuth 2.0 Token Exchange," RFC 8693, IETF, Jan. 2020.
- [13] B. Campbell, J. Bradley, N. Sakimura, and T. Lodderstedt, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens," RFC 8705, IETF, Feb. 2020.
- [14] D. Fett, B. Campbell, J. Bradley, T. Lodderstedt, M. Jones, and D. Waite, "OAuth 2.0 Demonstrating Proof of Possession (DPoP)," RFC 9449, IETF, Sep. 2023.
- [15] V. C. Hu et al., "Guide to Attribute Based Access Control (ABAC) Definition and Considerations," NIST Special Publication 800-162, Jan. 2014.

- [16] OpenAPI Initiative, "OpenAPI Specification v3.1.0," 2021.
- [17] OWASP Foundation, "Application Security Verification Standard 4.0.3," 2021.
- [18] Regulation (EU) 2016/679 (General Data Protection Regulation), Articles 5 and 25.
- [19] National Institute of Standards and Technology, "NIST Privacy Framework," version 1.0, Jan. 2020.
- [20] Istio Authors, "Authorization Policy," istio.io documentation.
- [21] Envoy Project Authors, "External Authorization," envoyproxy.io documentation.
- [22] OWASP Foundation, "Server-Side Request Forgery Prevention Cheat Sheet," cheatsheetseries.owasp.org.
- [23] N. Novaes Neto, S. Madnick, A. M. G. de Paula, and N. M. Borges, "A Case Study of the Capital One Data Breach," Cybersecurity at MIT Sloan, Working Paper, 2020.
- [24] T. Varygina and A. H. Bagge, "Overcoming Security Challenges in Microservice Architectures," in Proc. IEEE Symposium on Service-Oriented System Engineering (SOSE), 2018.
- [25] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing, "Automated Generation and Analysis of Attack Graphs," in Proc. IEEE Symposium on Security and Privacy, 2002.
- [26] A. Pereira-Vale, E. B. Fernandez, R. Monge, H. Astudillo, and G. Márquez, "Security in Microservice-Based Systems: A Multivocal Literature Review," Computers & Security, vol. 103, 2021.