Istio AuthorizationPolicy & HTTP/2 Coalescing

2026-07-13

Two Istio/Envoy pitfalls: AuthorizationPolicy IP matching (direct_remote_ip vs. remote_ip) and HTTP/2 connection coalescing across per-hostname listeners.

Isolating an admin surface behind its own hostname and an IP-restricted Istio `AuthorizationPolicy` sounds like a Tuesday-afternoon change: add a second `HTTPRoute`, add a `DENY` policy with an allowlist, done. Two non-obvious Envoy behaviors turned it into a two-day debugging exercise instead — one in how `AuthorizationPolicy` resolves "the client's IP," and one in how browsers reuse HTTP/2 connections across hostnames that share a certificate. Neither is specific to Keycloak or to this particular setup; both apply to any Istio or Envoy Gateway API ingress sitting behind a cloud load balancer with a shared wildcard certificate.

This post walks through both bugs, how they were root-caused, and the fix for each — so the next person hitting either symptom can skip straight to the cause.

Setup

The scenario: a public hostname (`auth.example.com`) serves an application's normal authentication endpoints, and a second hostname (`auth-admin.example.com`) is meant to serve the same backend's administrative surface, reachable only from a small IP allowlist. Both hostnames share one wildcard TLS certificate on the same Istio Gateway. The cluster sits behind a cloud load balancer that speaks the PROXY protocol, so Envoy can recover the real client IP instead of just seeing the load balancer's own TCP connection.

An `AuthorizationPolicy` with `action: DENY` and a negated IP match handles the allowlisting:

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: admin-restrict
spec:
  selector:
    matchLabels:
      gateway.networking.k8s.io/gateway-name: ingress-gateway
  action: DENY
  rules:
    - to:
        - operation:
```

```
        hosts: ["auth-admin.example.com"]
  from:
    - source:
        notIpBlocks:
          - "203.0.113.7/32"
```

`DENY` with negation, rather than an `ALLOW` policy, is deliberate: as soon as any `ALLOW` policy matches a workload selector, Istio flips that entire workload to default-deny for everything else — every other host on the same gateway would need its own explicit `ALLOW` rule too. A `DENY` policy with a negated match only affects the hosts it names.

Deployed as written above, this policy returned `403` for the allowlisted IP itself.

AuthorizationPolicy: direct_remote_ip vs. remote_ip

The first instinct when a `403` shows up for an IP that should be allowed is to check the CIDR: typo, wrong `/32` vs. `/24`, IPv4 vs. IPv6. None of that was wrong here. The next instinct was that `notIpBlocks` might not be the right field — Istio's `AuthorizationPolicy` API exposes three IP-matching options that look interchangeable: `source.ip`, `ipBlocks`/`notIpBlocks`, and `remoteIpBlocks`/`notRemoteIpBlocks`. Switching from `notIpBlocks` to `source.ip` with a `notValues` condition changed nothing — still `403` for the allowlisted IP.

That symptom — identical behavior across two supposedly different fields — is the tell that both resolve to the same underlying value. Confirming it required looking past the Istio CRD to what Envoy actually compiles the policy into, via a live config dump:

```
kubectl exec <istio-ingress-pod> -n <ns> -- \
  pilot-agent request GET config_dump?resource=dynamic_listeners
```

The compiled RBAC filter showed `direct_remote_ip` for the CIDR match — not `remote_ip`. Cross-checking against the Istio API proto source confirms it explicitly: `source.ip`, `ipBlocks`, and `notIpBlocks` all compile to Envoy's `direct_remote_ip`, described in Envoy's own docs as the address of the immediate TCP connection peer. Sitting behind a load balancer, that peer is the load balancer itself, not the browser that originated the request — the real client IP only reaches Envoy inside the PROXY protocol header (or an `X-Forwarded-For` entry), and only the `remoteIpBlocks`/`notRemoteIpBlocks` fields read that value (Envoy's `remote_ip`).

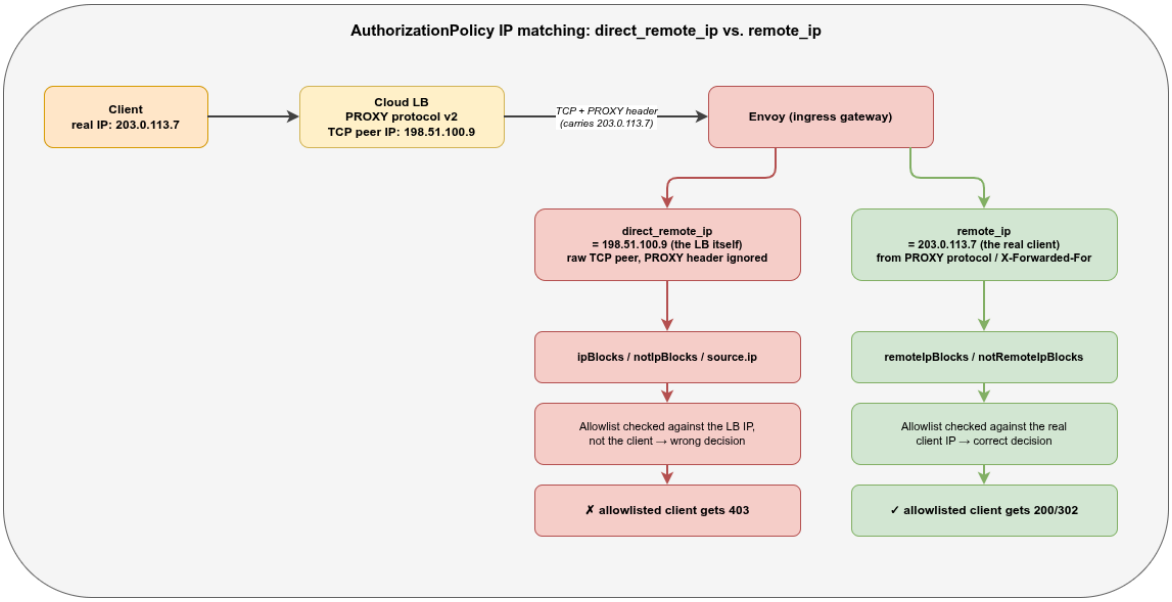

Figure 1: Diagram: an AuthorizationPolicy using ipBlocks or source.ip reads Envoy

The fix is a one-field swap:

```
    from:
      - source:
          notRemoteIpBlocks:
            - "203.0.113.7/32"
```

Verifying this live — before templating the fix — was done with `kubectl patch` directly against the running policy: swap the field, confirm the allowlisted IP stops getting `403`; then patch the CIDR list to a different, non-matching IP as a negative-control test and confirm `403` returns for the real client. Both checks passed before the change was committed to the source template.

Takeaway: `source.ip`, `ipBlocks`, and `notIpBlocks` in an Istio `AuthorizationPolicy` never see past the immediate TCP peer. Behind any proxy, load balancer, or PROXY-protocol listener, the only fields that read the actual client IP are `remoteIpBlocks`/`notRemoteIpBlocks`. When the two appear to produce identical results, that's a sign they're reading the same underlying field, not that the policy is correct.

HTTP/2 connection coalescing across per-hostname listeners

With the IP-matching fixed, the allowlisted IP stopped getting `403` — but the admin interface still failed to load in the browser, with a timeout on a background check the application ran against its own login-status endpoint. `curl` against the exact same URL, from the same IP, reliably returned `200`.

The Envoy access log for the failing request had two fields worth comparing:

```
authority: auth.example.com
requested_server_name: auth-admin.example.com
response_code_details: route_not_found
```

The request was addressed to `auth.example.com`, but its TLS SNI (`requested_server_name`) was `auth-admin.example.com` — a different hostname than the one actually being requested. The browser had opened its connection for `auth-admin.example.com`, then reused that same open HTTP/2 connection to request a resource on `auth.example.com` instead of opening a new one.

That behavior is spelled out in RFC 7540 §9.1.1: an HTTP/2 client is permitted to reuse an existing connection for a different origin whenever a certificate valid for both origins is available on that connection. A single wildcard certificate covering every hostname on the gateway makes that condition true for all of them — coalescing isn't a bug in the browser, it's specified behavior, and a `curl` invocation never triggers it because `curl` opens one connection per invocation and doesn't opportunistically reuse it across hostnames.

The Gateway in this setup had one Envoy listener per hostname — each with its own SNI-keyed filter chain and its own, independent route table. A request that arrives over a connection opened for a different hostname's filter chain finds no route for its actual target host in that filter chain's table, producing exactly the `route_not_found` seen in the log.

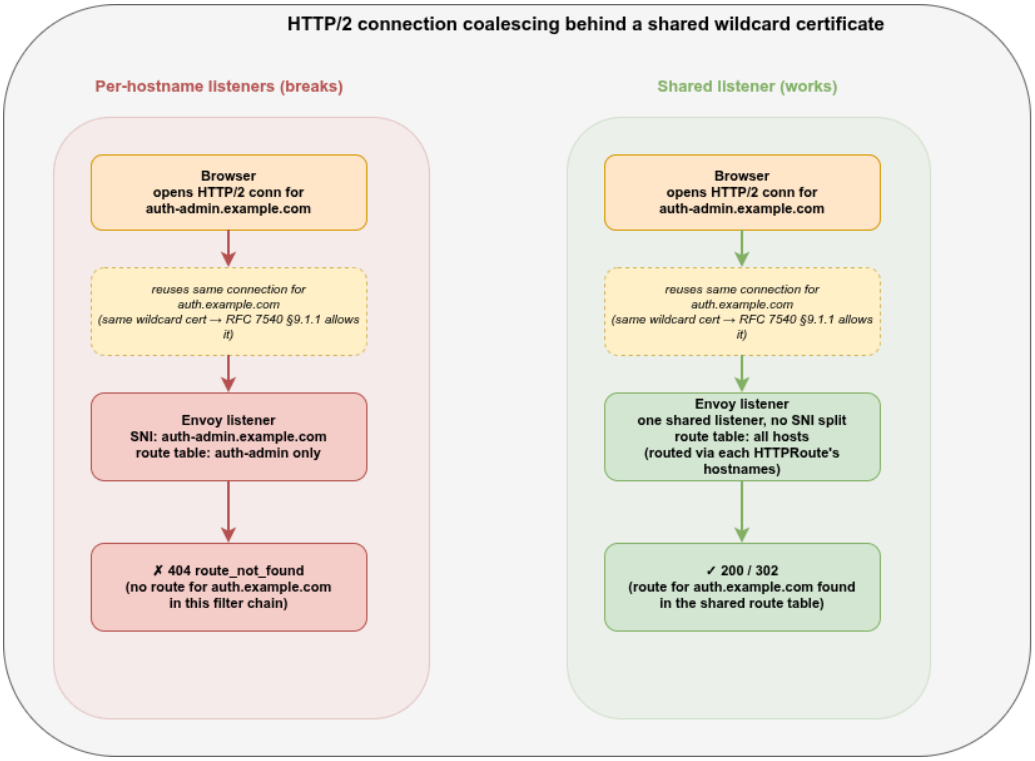

Figure 2: Diagram comparing two Istio Gateway listener setups: per-hostname listeners route a coalesced HTTP/2 request to the wrong filter chain and return 404 route_not_found, while a single shared listener routes the same coalesced request correctly via HTTPRoute hostnames

The fix collapses the per-hostname listeners into a single shared listener on port 443, with no `hostname` field restricting it to one SNI value:

```
listeners:
  - name: https
    port: 443
```

```
    protocol: HTTPS
    tls:
      mode: Terminate
      certificateRefs:
        - name: wildcard-tls
    allowedRoutes:
      namespaces:
        from: All
```

Host-based routing keeps working exactly as before, because it was never actually tied to the listener count — each `HTTPRoute`'s own `hostnames` field is what Envoy uses to route a request to the right backend, regardless of which listener or SNI value the connection came in on. One shared listener means one shared route table, so a coalesced request finds the right route no matter which hostname's connection carried it in.

Takeaway: any Gateway API setup with more than one hostname behind a single certificate — a wildcard cert is the common case — should route those hostnames through one shared listener, not one listener per hostname. Per-hostname listeners work fine until a client coalesces two of them onto the same connection, at which point routing breaks for exactly the requests that happen to travel over a reused connection — intermittent-looking, browser-dependent, and invisible to `curl`.

Takeaways for your own Istio ingress

- If an `AuthorizationPolicy` IP allowlist denies the IP it's supposed to allow, check which field is doing the matching before checking the CIDR. `remoteIpBlocks`/`notRemoteIpBlocks` are the only fields that see past a load balancer or reverse proxy to the real client IP.
- When in doubt about what an `AuthorizationPolicy` actually compiles to, read Envoy's own `config_dump` rather than inferring behavior from the CRD field names alone — `ipBlocks` and `remoteIpBlocks` sound like they should differ only in strictness, not in which network hop they inspect.
- Multiple hostnames sharing one certificate on the same Gateway are candidates for HTTP/2 connection coalescing. A per-hostname Envoy listener setup will misroute exactly those coalesced requests — test with a real browser, not only `curl`, before ruling this out.
- A `DENY` policy with a negated match, rather than an `ALLOW` policy, avoids collaterally flipping every other host on the same Gateway to default-deny.

Summary

Two Istio/Envoy behaviors that look correct in the YAML and still produce the wrong result in practice: `AuthorizationPolicy` IP fields that silently match the wrong network hop, and HTTP/2 clients that silently reuse connections across hostnames sharing a certificate. Both were only visible by comparing Envoy's actual compiled configuration and access logs against what the higher-level API implied — the RBAC decision and the routing decision each happen one layer below where the policy is written.