ALTSECCON 2026

One Key to Sign Them All

Forging Service Identity at the Mesh Boundary

Sweta Mantraratnam

About Me



Sweta Mantraratnam

Security Architect

12+ years securing systems at scale

Cloud security by day, bug hunter by nature

Loves whiteboarding designs down to first principles

DISCLAIMER

The views, content and opinions expressed in this presentation are entirely my own and
do not represent, reflect or constitute the views, policies or positions of my employer.

Are you running a service mesh ?

Are you relying on mesh controls for Authentication and Authorization?

You are probably securing identity with ...

Strict mTLS, AuthorizationPolicy,

RequestAuthentication, **SPIFFE identities,**

PeerAuthentication: STRICT, JWT validation,

NetworkPolicies, **a zero-trust posture,**

...

Is that enough?

No.

Provenance + Custody *decide the outcome – not mTLS.*

ACT 1 · THE IDENTITY PLANE

How mTLS carries identity

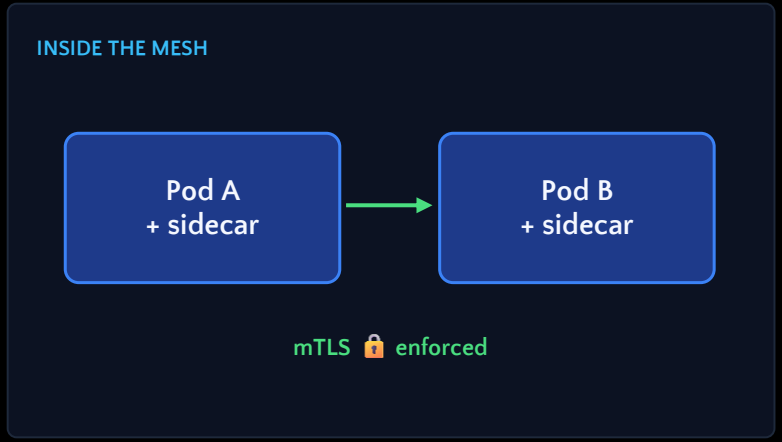
Every workload gets a SPIFFE identity

trust domain + namespace / service account, in a short-lived cert.

```
spiffe://mesh.internal/ns/app1namespace/sa/app1
```

The mesh enforces mTLS sidecar-to-sidecar.

Strong - but only on the wire, between two sidecars, in one trust domain.



ACT 1 · THE IDENTITY PLANE

… and where it dies

mTLS identity is not portable. It cannot survive a TLS hop - a load balancer, an ingress, a hop to another cluster / trust domain. At each TLS-terminating boundary the peer certificate is stripped.

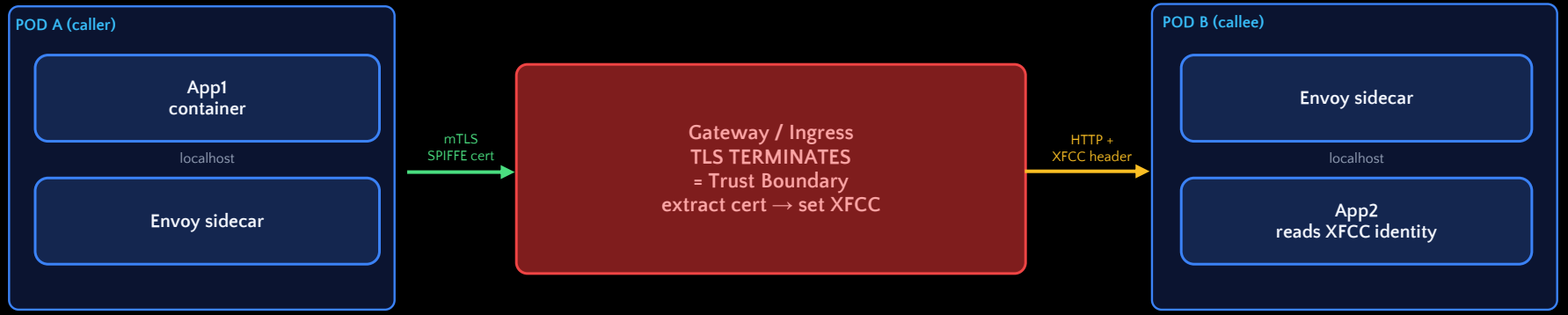


The cert does not travel with the request. **Whatever re-materialises identity on the far side is now your entire trust model.**

ACT 1 · THE IDENTITY PLANE

① XFCC – cert identity becomes a header

App1's mTLS / SPIFFE identity is valid only sidecar-to-sidecar. At the TLS-terminating boundary the peer cert is stripped and its identity is copied into the XFCC header – App2 now trusts a header, not a cert.



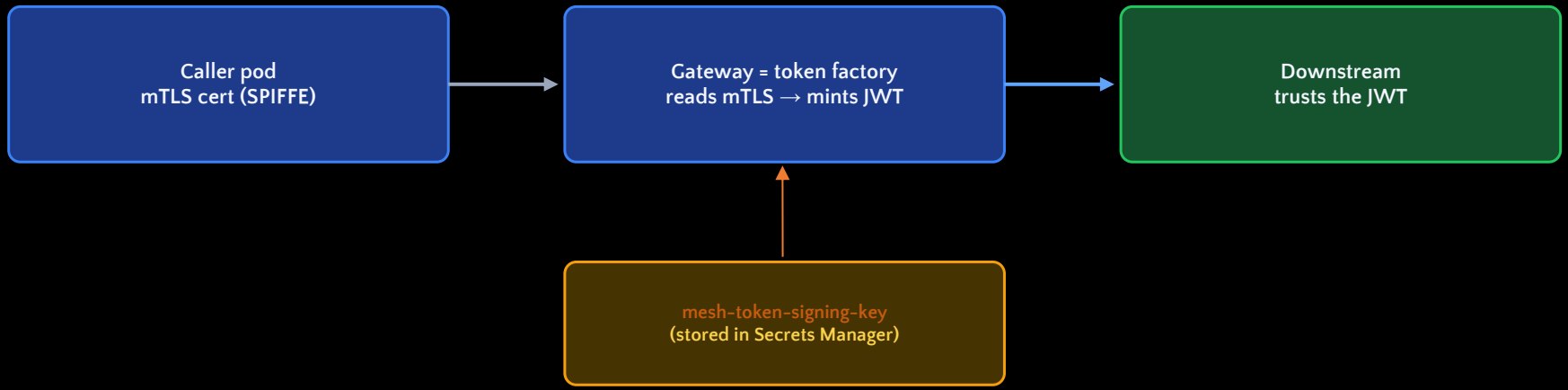
spiffe://mesh.internal/ns/app1namespace/sa/app1

The peer certificate is stripped at the boundary – it does NOT travel to App2. Whatever sets XFCC after termination is the trust model – unless the gateway re-derives identity from the cert.

ACT 2 · THE NEAR-UNIVERSAL FIX

The gateway becomes a token factory

To carry identity across the boundary, a gateway reads the caller's real mTLS cert and mints a signed JWT – using one RSA key in a cloud secret manager.

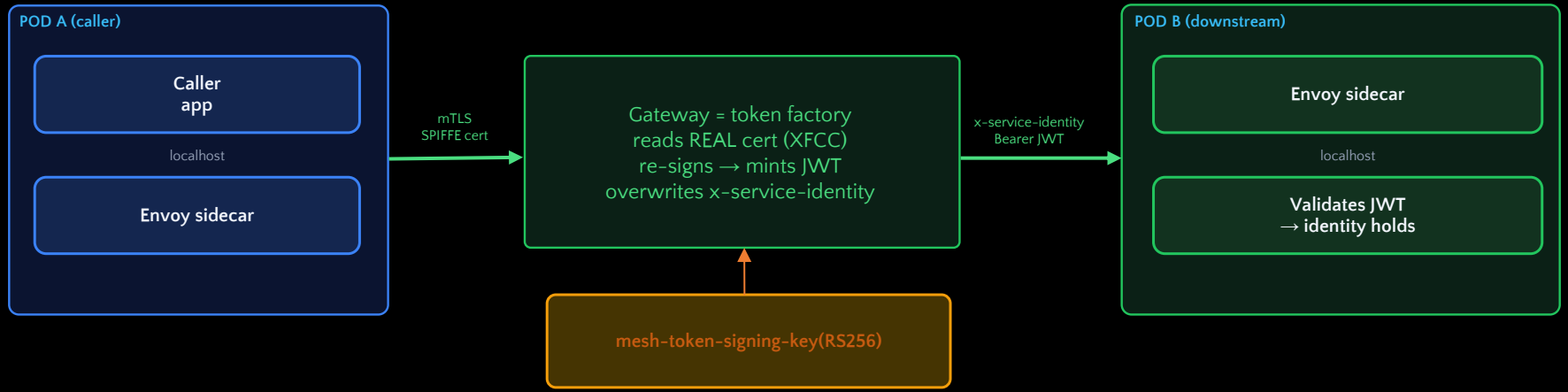


The entire trust model now rests on one recoverable key. That is the thread we pull.

ACT 2 · ATTACKING THE RE-SIGNER

② x-service-identity – re-signed from the cert

The gateway ignores any attacker-supplied token, re-derives identity from the REAL mTLS cert, and re-signs a fresh JWT with one signing key. Provenance = cert → forgery fails.

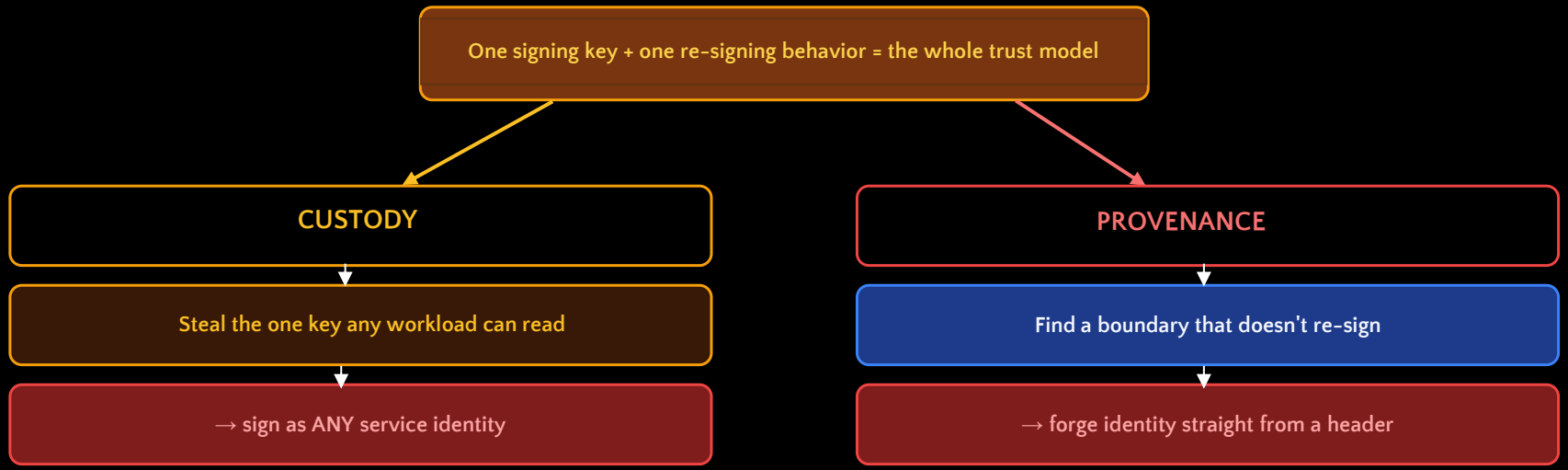


Any x-service-identity the caller supplies is discarded. **Identity is re-derived from the cert and re-signed – a stolen key alone can't forge it here.**

ACT 2 · PULL THE THREAD

One key signs for everyone – so what breaks?

Everything from here is one of exactly two failures. Hold both paths – every act that follows pays one of them off.

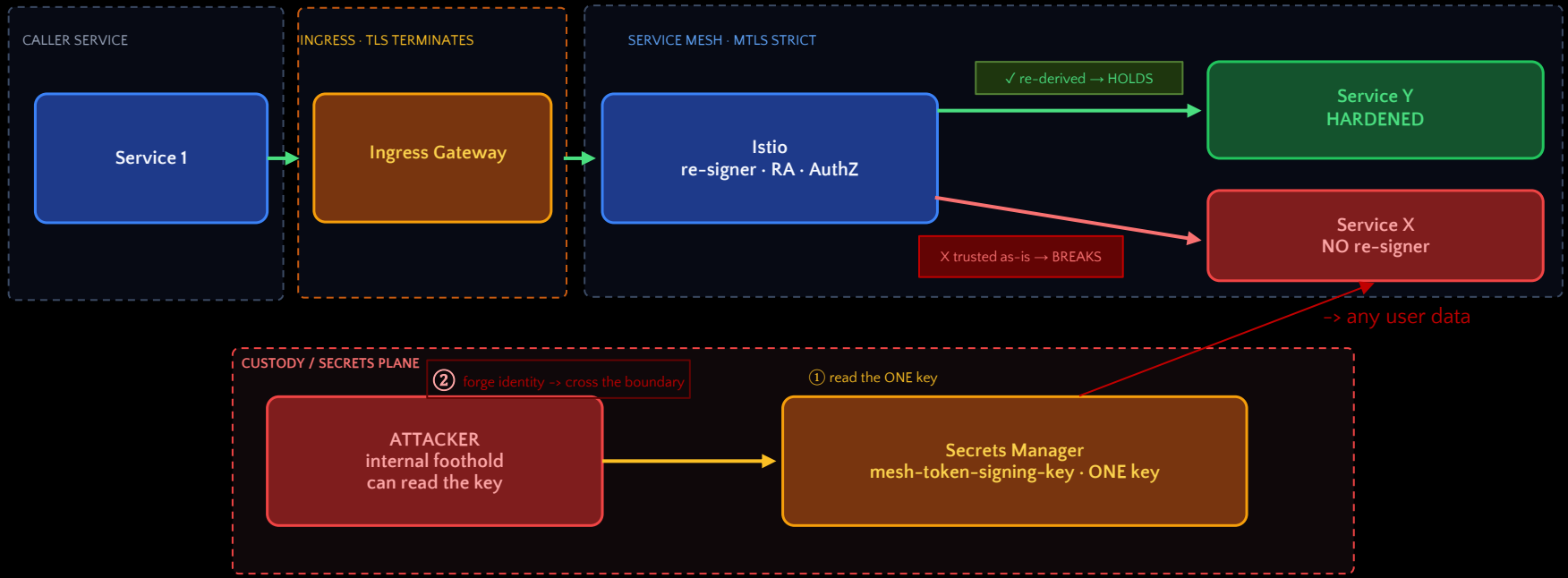


Recon proves how broad custody really is

The service X path is exactly this failure

ACT 3 · TRUST BOUNDARIES

The system – and where trust breaks



dashed = trust boundary green = holds (re-signer + RA + AuthZ) red = attacker / break

ACT 3 · THREAT MODEL

Two attack classes, mapped to STRIDE

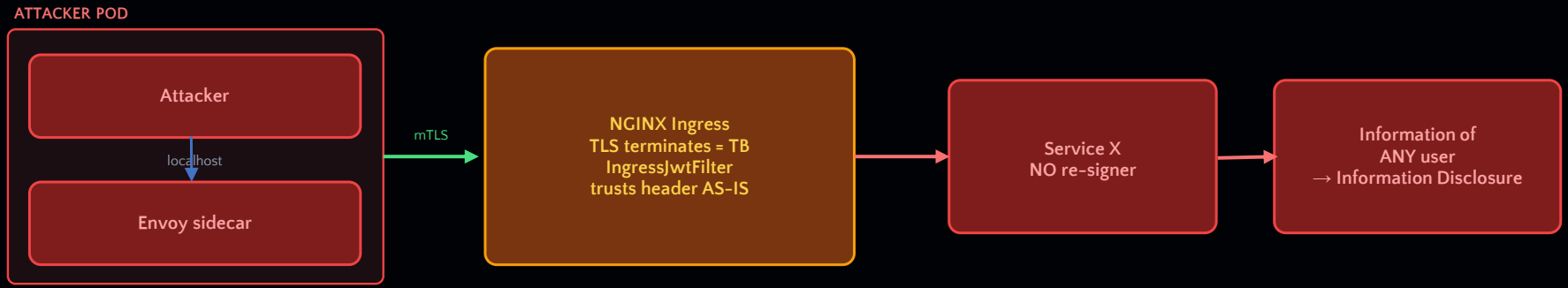
S	**Spoofing**	Forge a service identity across the boundary.
E	**Elevation of privilege**	Impersonate a privileged service to reach a guarded endpoint.
I	**Information disclosure**	Leak sensitive information values in the response.

The deciding variable was never the traffic direction or cluster hops - **it was whether a re-signer sits on the boundary.**

ACT 3 · THE RE-SIGNER MISSING

③ Service X path – header trusted as-is

The service X sits behind an ingress that trusts the interservice JWT as supplied – no re-signer, no caller↔target binding.



Forged sub=serviceX, POST /app1serviceX → 200 OK { sensitiveinfo: "personaladdress" } for any victim's ID, in cleartext.

ACT 3 · CUSTODY RECON

Who can read the one key?

```
$ ./enum-secret-readers.sh 'mesh-token-signing-key'

GetSecretValue ✓  alt-gateway-role        # intended
GetSecretValue ✓  any-reader-lambda       # broad
GetSecretValue ✓  vault-service-ba         # broad
GetSecretValue ✓  service1-gw-lambda       # intended
GetSecretValue ✓  service2-*-svc   (x32)   # unexpected
GetSecretValue ✓  dev-1-svc   (x9)         # unexpected

(!) Multiple principals can read the signing key.
    Only 2 should.
```

CUSTODY INVARIANT

A signing key any workload can read = any workload can mint any identity.

Recon = two joins

1. IAM principals with GetSecretValue on the key.
2. Which of those pods can actually reach the endpoints.

ACT 3 · ATTACKING THE RE-SIGNER

Where a re-signer sits, forgery dies

1 Inject a claims header

DEFEATED – RequestAuthentication overwrites it with verified claims.

2 Forge a signed JWT with the key

DEFEATED – Gateway re-signs from the real cert – identity collapses back to our pod.

3 Impersonate cross-namespace

DEFEATED – AuthorizationPolicy denies it before app code runs.

```
$ curl -i .../serviceX/app1 \
  -H "x-service-identity: Bearer
<forged
     sub=serviceX-appX>"

HTTP/1.1 403 Forbidden
x-istio-principal:
  spiffe://mesh.internal/ns/attacker/
  sa/our-real-pod
          # <- forged sub gone
```

The resolved identity in the response is our REAL pod – not the service we forged.

ACT 4 · THE REAL BREAK

One endpoint has no re-signer

The service X service is reached through an ingress that trusts the interservice JWT as supplied. We audited its Java filter line-by-line.

```
// IngressJwtFilter - trusts headers
boolean viaIngress =
    xfcc.contains("ingress-PRINCIPAL")
    && host.startsWith("svc-serviceX-svc");

// identity from a header, never the cert:
sub = headers.get("x-service-id-claims").sub;

if (!sub.equals(EXPECTED_SUB))
    return UNAUTHORIZED;
```

GAP 1

No re-signer. Identity is read from x-service-id-claims, never re-derived from the mTLS cert.

GAP 2

Substring XFCC match -contains(), no field/boundary parsing of the client-cert header.

GAP 3

Prefix host match - startsWith() - accepts crafted x-forwarded-host values.

ACT 4 · THE MONEY SHOT

One signed request → any customer's information

```
$ curl -s -i https://app1-internal-svc.mesh.abc.io/v1/serviceX \
  -H "x-service-identity: Bearer eyJhbGci...<forged sub=serviceX-gw-...>" \
  -H "Content-Type: application/json" \
  -d '{"value":"137376..<ID>","channel":"API"}'

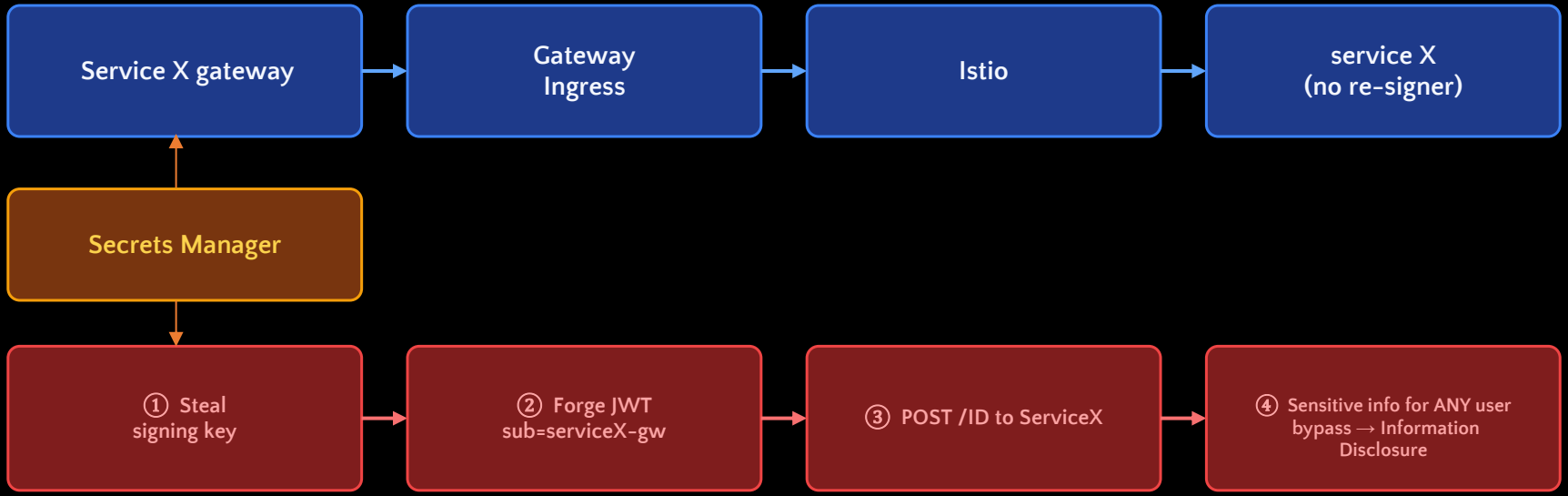
HTTP/1.1 200 OK
content-type: application/json

{ "sensitiveinfo": "173 Prime Walk EXX 2XX" }
```

channel=API returns the user's address in the response body. **No re-signer, no caller/target binding - we asked for any victim's ID and got their address in cleartext.**

ACT 4 · THE CHAIN

The Information disclosure chain



blue = legitimate flow · amber = shared secret · red = attacker path

DEMO

```
sm@demo demo-lab % kubectl get pods -n attacker
NAME                        READY   STATUS    RESTARTS   AGE
attacker-696b4db5c6-qhhbl   2/2     Running   0          146m
sm@demo demo-lab %
```

ACT 5 · BEYOND THE MESH

Do you know what's in and out of mesh?

mTLS and AuthorizationPolicy only govern traffic that goes through the sidecar. Workloads can quietly leave the mesh.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    traffic.sidecar.istio.io/
      excludeInboundPorts: "8080"

# traffic to :8080 never hits the sidecar
# no mTLS   no AuthZ   no RequestAuthentication
```

Is this a blind spot?

"I have no idea how you'd monitor this at scale." A pod outside the mesh is invisible to every mesh-level control you rely on.

Does anything govern it today?

No policy blocks it out of the box. Add Kyverno / OPA to flag un-injected pods and excludeInboundPorts usage - one signal for "pods outside mesh."

What happens?

With a permissive PeerAuthentication / DestinationRule, mesh traffic can fall back to plaintext. Worth a deep-dive on your own config.

ACT 5 · BEYOND THE MESH

The bypass: cluster-admin

Default ClusterRoles like cluster-admin (system:masters) lets a user tunnel straight to a pod - around every network-level control.

```
$ kubectl auth can-i '*' '*'      # system:masters
yes
$ kubectl port-forward pod/serviceX 9000:8080
Forwarding from 127.0.0.1:9000 -> 8080

# direct tunnel to the pod:
#  bypasses AuthorizationPolicy
#  bypasses NetworkPolicy
#  the serviceX filter is all that's left
```

Why it matters

AuthorizationPolicy + NetworkPolicy are exactly the controls teams lean on for in-cluster authz.

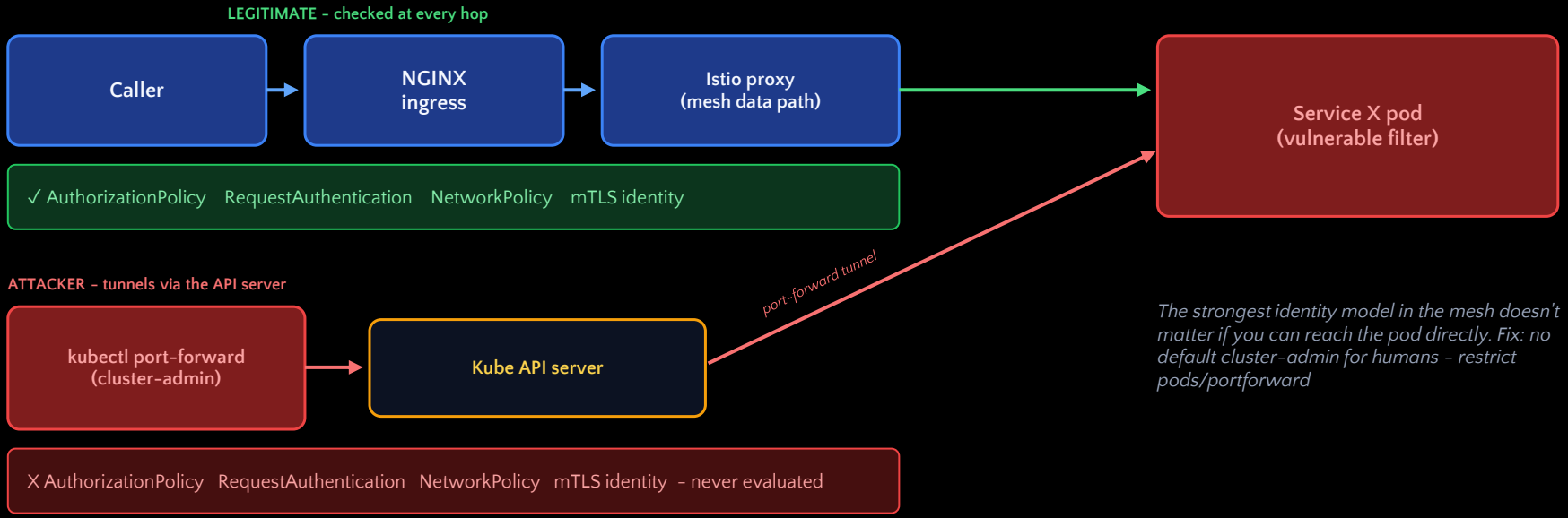
port-forward goes under both. A high-privilege human - or a stolen kubeconfig - skips the entire identity model.

Fix: no default cluster-admin for humans; audit / restrict pods/portforward

ACT 5 · THE HUMAN BYPASS

port-forward: a tunnel around the mesh

cluster-admin (system:masters) can tunnel straight to a pod through the API server and the request never enters the data path the mesh guards.



The strongest identity model in the mesh doesn't matter if you can reach the pod directly. Fix: no default cluster-admin for humans - restrict pods/portforward

THE FRAMEWORK

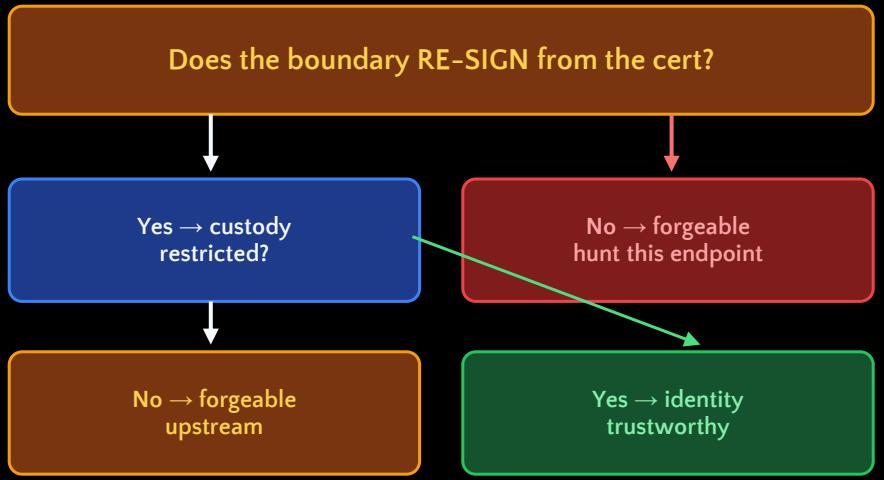
Two questions decide every outcome

PROVENANCE

Re-derived from the cert at each boundary → forgery fails. Trusted from a header → forgery wins.

CUSTODY

A signing key any workload can read means any workload can mint any identity.



DEFEND

How each fix flips the chain – red to green

Every attacker capability, and the one control that turns it from red to green.

ATTACKER CAN…	APPLY THIS CONTROL	…AND IT'S GREEN
Read the shared signing key	per-service keys + tighten IAM	✓ no one key mints every identity
Spoof identity on the service X path	re-sign from the cert + RequestAuthentication	✓ forged claims overwritten at the boundary
Request any user's address	bind target to caller session + exact-match XFCC / host	✓ a caller can only act for itself
Tunnel around the mesh	govern membership + kill default cluster-admin	✓ no path skips the sidecar

Note what's missing: "rotate the key" isn't on the list. *The key was never the point – the missing re-signer was.*

TAKEAWAYS

Takeaways

1

Fingerprint the boundary

Does it re-sign from the cert? That one test decides whether forgery is possible at all.

2

Hunt the 2nd token factory

The takeover lives in the auth-adjacent service OUTSIDE the re-signer / RequestAuthentication.

3

Provenance + Custody

Re-derive identity at every boundary; restrict who can hold the key - and who can bypass the mesh.

Mesh mTLS is necessary but not sufficient *- identity is only as strong as its weakest re-signing boundary and its broadest key custody.*

Q & A

Design Patterns and demo lab available on Github -> @smantra