Your CI's Mistaken Identity

Task-Scoped Trust in Cloud-Native Pipelines

Andrew McNamara (Red Hat) • Maia Iyer (IBM Research)



KubeCon + CloudNativeCon North America · November 10, 2026

The Hook: A Simple Question

> **"You wouldn't ask a plumber to sign off on your electrical work."**

Yet in nearly every Kubernetes CI/CD pipeline running today:

- One shared `ServiceAccount` runs every step in the pipeline.
- One ambient cloud credential or key signs SBOMs, scans, and release claims.
- A compromised dependency-fetch task inherits the **exact same authority** as build and release tasks.

> **The Problem:** Workload identity is only as granular as you configure it. In most CI/CD deployments it's configured around *location* (namespace + ServiceAccount) rather than *role* (what the workload actually is) — leaving downstream services to infer authority from a signal that doesn't carry it.

The Identity Perspective: Why the Standard Deployment Model Fails Us

In the cloud-native identity world, we solved machine-to-machine authentication with **Workload Identity (SPIFFE/SPIRE)**.

We said: *"No more hardcoded API keys or static credentials in Kubernetes Secrets."*

The Standard SPIFFE Deployment Model:

Workload Attestors inspect the running container:

- Kubernetes Namespace (`default-tenant`)
- ServiceAccount Name (`pipeline-runner`)
- Node or Cluster Name (`prod-cluster-01`)

```
spiffe://company.org/ns/default-tenant/sa/pipeline-runner
```

The Blind Spot: Ephemeral CI vs. Microservices

SPIFFE was designed with **long-running microservices** in mind:

- **Microservices Model:**
  - Predictable, homogeneous workloads (`service-a` talking to `service-b`).
  - The ServiceAccount boundary *is* the application boundary.
  - Location implies identity.
- **CI/CD Pipeline Reality:**
  - Ephemeral DAGs of wildly heterogeneous tools.
  - Untrusted PR checkouts, third-party package downloaders, compiler toolchains, scanners, and release signers all run under **one tenant identity**.

> **In CI/CD:** ServiceAccount ≈ The entire factory. Location alone no longer tells you who is calling.

The Supply Chain Threat: Confused Deputies in CI

When identity is coarse, every pipeline step becomes a potential **Confused Deputy**:

```
PipelineRun: build-and-test
  ├── Task 1: git-clone         [SA: runner]  ◄── Untrusted code / PRs
  ├── Task 2: fetch-deps        [SA: runner]  ◄── Dynamic package downloads
  ├── Task 3: build-container   [SA: runner]  ──► Needs builder authority
  └── Task 4: trivy-scan        [SA: runner]  ──► Needs scanner authority
```

- All four tasks share `sa/runner` in `default-tenant`.
- Early steps **execute untrusted code by design** — PR code, dependency install scripts — so a compromised Step 1 or 2 runs with the full authority of Step 3 and Step 4.
- Provenance + policy verification catches a forged *artifact* — but the hijacked step's authority abuse (secrets, tag pushes, service access) happens outside the provenance channel, before any verifier sees it.

Anatomy of an Attack: Ambient Authority Hijack

How an attacker exploits shared namespace credentials:

1. **Compromised Dependency / PR:** Attacker injects code during `git-clone` or `fetch-deps`.
2. **Ambient Credential Access:** Malicious script reads the mounted `regcred` secret or projected ServiceAccount token.
3. **Identity Impersonation:** Because all tasks share `sa/runner`, the compromised step possesses push and sign privileges.
4. **Forged Integrity:** Attacker overwrites production tags or signs false claims: *"0 vulnerabilities, clean SBOM"*.

> **The Fundamental Gap:** The Pod ran with ambient cluster authority. The cluster had no verification of what code was executing inside the container.

Live Demo: Act 1 — The Breakdown (Ambient Authority & Secret Hijack)



Part 2: Building Task Identity

Tekton Resolvers, Admission Control, and SPIRE

Tekton Architecture: Tasks, TaskRuns, and Resolvers

How do pipelines execute in cloud-native Kubernetes?

- **`Task`** : Reusable definition containing container steps (commands, env vars, image).
- **`TaskRun`** : Execution instance that instantiates a Kubernetes `Pod` to run the steps.
- **`Pipeline`** & **`PipelineRun`** : Directed Acyclic Graph (DAG) orchestrating multiple TaskRuns.

```
TaskRun ──creates──► Pod        (labeled tekton.dev/task)
  Step  ──maps────►  Container  (image + command defined by the Task)
```

The Problem: Where Do Tasks Come From?

- Historically, tasks were pasted inline into YAML or installed cluster-wide.
- If developers write inline bash in `TaskRun.spec.taskSpec`, they can execute arbitrary, unvetted binaries.
- **Tekton Resolvers**: Remote, pluggable task resolution (`git`, `cluster`, and `bundles`).

Tekton Resolvers: Pinned OCI Bundles

A **Tekton Bundle** is a Tekton `Task` packaged inside an OCI container image layer:

- The bundle pins the **exact container images, commands, and scripts** each step will run — verifying the bundle verifies the code **before any Pod is scheduled**.

```
apiVersion: tekton.dev/v1
kind: TaskRun
metadata:
  generateName: demo-signed-task-
  namespace: default-tenant
spec:
  taskRef:
    resolver: bundles
    params:
    - name: bundle
      value: registry.kind/tekton-catalog/builder@sha256:ce7492...
    - name: name
      value: buildah-oci-ta
```

Why OCI Bundles Change the Game

Task definitions become verifiable *before execution*:

1. **Immutable by Digest**
   - A TaskRun pins the exact task definition (`@sha256:...`).
   - The cluster knows *precisely what code will run* — no tag drift, no silent mutation.
2. **Signable like any OCI Artifact**
   - Stored in existing registries, signed with Cosign / Fulcio keyless.
   - "Trusted catalog" = a signing key, not a curated git repo.
3. **Visible at Admission — Before Any Pod Exists**
   - `taskRef.params.bundle` is inspectable by webhooks the moment the TaskRun is submitted.
   - Verify signature → classify role → *then* the Pod is created. Unverified code never receives a **trusted** identity — it runs, but identified only as what it was verified to be.

Kyverno at the Gate: Admission-Time Verification

When a `TaskRun` is submitted, Kyverno intercepts it **before any Pod is scheduled** — the policy matches on `tekton.dev/taskruns`, then narrows via CEL to bundles matching the **trusted catalog patterns**:

1. **Extract** each pinned bundle digest from `taskRef.params`.
2. **Verify**: catalog bundles must be `@sha256:`-pinned **and** signed by the trusted catalog key.
3. **Decide**: catalog bundle failing either → **Deny**. Everything else (inline scripts, git tasks, unknown registries) is *not blocked* — it just **never gets promoted**.

```
kind: ImageValidatingPolicy        # Kyverno (CEL), matched on tekton.dev/taskruns
spec:
  images:
  - name: taskBundle
    expression: 'object.spec.taskRef.params.filter(p, p.name == "bundle").map(p, p.value)'
  validations:
  - expression: 'images.taskBundle.all(img, img.matches("@sha256:[a-f0-9]{64}$"))'
  - expression: 'images.taskBundle.map(img, verifyImageSignatures(img, [attestors.catalogKey])).all(v, v > 0)'
```

> **Reading it:** catalog bundles must be digest-pinned and signed by `attestors.catalogKey` (the trusted catalog's public key) — one failure denies the whole TaskRun.

Policy Enforcement: Dynamic Role Promotion

How admission policy categorizes tasks before execution:

- **Verified Signed Bundles**
  - Kyverno mutates the TaskRun label: `trusted-task-role: prod`.
- **Everything Else** — inline scripts, git-resolved tasks, unknown registries
  - Every TaskRun is labeled `trusted-task-role: dev` **by default**; trust is opt-in, never assumed.
- **Anti-Spoofing Policy (`prevent-pod-label-spoofing`)**
  - Blocks pods from self-assigning or altering `trusted-task-role` labels.

> **Result:** Pod labels reflect cryptographically verified admission state, immune to tenant pod tampering.

Step 2: SPIRE Mints the Role SVID

`ClusterSPIFFEID` evaluates the Kyverno-verified pod labels:

```
apiVersion: spire.spiffe.io/v1alpha1
kind: ClusterSPIFFEID
metadata:
  name: konflux-trusted-prod
spec:
  spiffeIDTemplate: >-
    spiffe://konflux-ci.dev/trusted/kind-konflux/{{ .PodSpec.ServiceAccountName }}/{{ index .PodMeta.Labels "tekton.
  podSelector:
    matchLabels:
      trusted-task-role: "prod"
```

- SPIRE node agent attests the Pod against the Kubernetes API.
- Only Pods possessing the verified `trusted-task-role: "prod"` label receive the production SVID template.

Cryptographic Task Roles in Practice

Three tasks sharing `sa/default`, but minted distinct cryptographic identities:

```
# Scanner Task (Verified Catalog Bundle)
spiffe://konflux-ci.dev/trusted/kind-konflux/default/trivy-sbom-scan

# Builder Task (Verified Catalog Bundle)
spiffe://konflux-ci.dev/trusted/kind-konflux/default/buildah-oci-ta

# Untrusted Task (Inline Script or Unsigned Bundle)
spiffe://konflux-ci.dev/dev/kind-konflux/default/arbitrary-script
```

> **Key Architectural Leap:** All three tasks share `sa/default`, but receive completely different cryptographic identities based on *admission-verified task code*!

Live Demo: Act 2 — Task Admission & Cryptographic Identity

Live Demo: Act 2

ttyd offline

Task Admission Classification & SPIRE Identity Minting

- Inline untrusted tasks are classified into the unprivileged `dev` role.
- Attacker attempting to spoof catalog namespace with unsigned task is denied at admission.
- Cosign-signed catalog bundle verified by Kyverno, promoted to `prod`, and minted production SVID.

```
./demo/run-demo.sh --act 2
```

Copy

To run live in slide: `./demo/serve-slides.sh start` · No demo terminal? Run locally: `git clone github.com/arewm/slsa-konflux-example` (branch `kubecon-na-2026-your-cis-mistaken-identity`), then the same command — see `demo/README.md`

Retry Connection

Part 3: What This Unlocks

Location vs. Authorization & Same-Namespace API Gating

The Implications: Breaking the Location Trap

What did we just establish?

1. We Broke the "Location = Authorization" Trap

- In microservices, knowing `default-tenant:default` was enough because the service was single-purpose.
- In CI/CD, location alone is meaningless.
- We now have **cryptographic proof of what code is executing**, certified by admission control.

2. We Preserved SPIFFE's Core Tenets

- **SPIFFE does Authentication**: Cryptographically asserts *who* the workload is.
- **Consuming Systems do Authorization**: Decides *what* that workload is allowed to do.
- We didn't mutate SPIFFE—we fed SPIRE verified admission metadata!

> **The Principle:** Authorization must follow *Role*, not *Location*.

Generalizing Within the Namespace: Three Core Patterns

Once tasks possess fine-grained cryptographic identities, how do we use them inside the tenant namespace?

Pattern 1: Federated OCI Push Gating
Eliminate ambient registry secrets (`regcred`). Configure the OCI registry with OIDC Bearer auth matching SPIFFE SVIDs—only vetted builder tasks can initiate push sessions.

Pattern 2: Portable Secretless Service Access
Internal microservices (CVE databases, KMS, artifact stores) validate caller SVIDs directly against SPIRE's `/keys` JWKS endpoint without mounting static Kubernetes Secrets.

Pattern 3: Separation of Duties Attestations
Tasks sign in-toto claims via Sigstore/Fulcio. Policy engines (Conforma / OPA) verify that the certificate SAN matches the authorized role (scanners sign CVEs, builders sign SBOMs).

Pattern 1: Federated OCI Push Gating

Zot OCI registry validates bearer JWTs against SPIRE's OIDC discovery endpoint (`/keys`):

```
{
  "repositories": {
    "slsa-e2e-test": {
      "policies": [{
        "users": ["spiffe://konflux-ci.dev/trusted/kind-konflux/default/buildah-oci-ta"],
        "actions": ["read", "create"]
      }]
    }
  }
}
```

- **Dev / Attacker Task SVID →** Rejected with `HTTP 403 Forbidden`.
- **Vetted Builder Task SVID →** Accepted with `HTTP 202 Accepted`.
- **Zero ambient push tokens** stored in the tenant namespace.

Pattern 2: Portable Secretless Service Access

Internal cluster services authenticate callers using SPIRE JWTs directly:

- **Zero Static Secrets:** No API keys or database tokens mounted into pipeline pods.
- **Direct JWKS Validation:** Services validate bearer JWT signatures against SPIRE's `/keys` endpoint.
- **Role-Based Authorization:**
  - Only `default/trivy-sbom-scan` is permitted to query the CVE vulnerability feed.
  - Inline tasks or unauthorized builders receive `HTTP 403 Forbidden`.
- **Legacy Services Without OIDC:**
  - A broker (e.g. Vault) trusts SPIRE's OIDC endpoint and exchanges a JWT-SVID for a static credential — issued only to specific SPIFFE IDs.
  - The pod still holds **no long-lived secret**; the exchange is per-run and role-scoped.

> **Benefit:** Secrets cannot leak from compromised containers because the secrets simply do not exist in the tenant pod.

Pattern 3: Scoped Attestations & Separation of Duties

Pipelines produce multiple in-toto attestations, each signed by a distinct role:

- **Builder Task (`buildah-oci-ta`):**
  - Authorized to sign container images and SBOMs (`https://spdx.dev/Document/v2.3`).
- **Scanner Task (`trivy-sbom-scan`):**
  - Authorized to sign vulnerability scans (`https://aquasecurity.github.io/trivy/report/v1`).

The Threat:

What happens if a compromised build step generates a false vulnerability report claiming: *"Zero CVEs Found"*?

Enforcing Separation of Duties with Conforma (OPA Rego)

Conforma verifies that the X.509 certificate SAN URI matches the authorized role:

```
package policy.cve

# Rule: Vulnerability reports MUST be signed by an authorized scanner
deny[msg] {
  attestation := input.attestations[_]
  attestation.predicateType == "https://aquasecurity.github.io/trivy/report/v1"

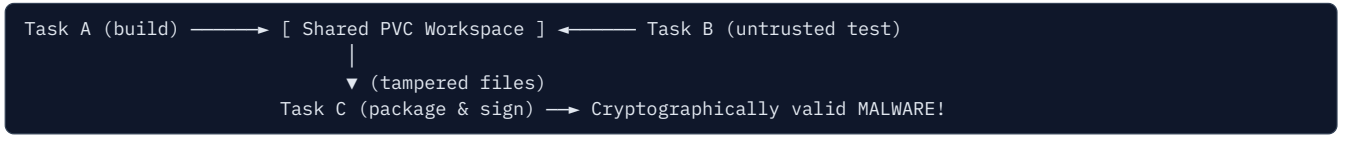
  not startswith(attestation.certificate.uri,
                 "spiffe://konflux-ci.dev/trusted/kind-konflux/default/trivy-sbom-scan")

  msg := sprintf("CVE scan signed by unauthorized role: %s", [attestation.certificate.uri])
}
```

> **The Guarantee:** Even if a builder signs a zero-CVE report, Conforma rejects it because the builder is not an authorized scanner!

What Identity Alone Doesn't Solve: The Data Plane

Does securing task identities protect the pipeline? **Not if tasks share a disk.**



OCI Trusted Artifacts close the loophole: each task packages its output into an immutable, digest-pinned OCI layer; downstream tasks fetch strictly immutable digests, never shared mutable volumes (required for SLSA Build Level 3).

> **Key Principle:** Control plane identity (SPIRE) + Data plane immutability (OCI Trusted Artifacts) = End-to-End Pipeline Integrity.

Part 4: The Next Frontier

The Managed Release Boundary

The Managed Release Boundary

Dual-gating. The release SVID is minted only when **three conditions all hold**:

1. An operator-designated **release namespace**
2. The **one sanctioned release pipeline** — classified at admission time
3. The **designated signing task** within it

Every other step in that pipeline — including the steps *before* policy verification — is denied an identity entirely. Keyless-signed with Cosign into Rekor.

```
spiffe://konflux-ci.dev/release/demo-app/slsa-e2e-release-dual-gated
```

Guaranteeing policy before privilege. Dual-gating still *assumes* the pipeline orders verification before signing. To close that: bind the privileged action to the policy's success. On a passing run, the policy task materializes a **short-lived, run-scoped signing capability** that only the sanctioned signer *in that same run* can obtain — no passing policy, no capability, no signature. "Policy passed" becomes a **verified fact**, not a pipeline-ordering assumption.

> From *structural* trust to *cryptographic* trust. Find out more in the demo repo.

Key Takeaways

1. **Location ≠ Role**: Stop treating Kubernetes namespaces and generic ServiceAccounts as authorization boundaries.
2. **Shift Verification to Admission**: Intercepting task bundles at admission with Kyverno prevents untrusted code from ever gaining production identities.
3. **Task-Scoped Identity is Real Today**: Combining Tekton, Kyverno, SPIFFE/SPIRE, and Sigstore brings least-privilege cryptographic identity to every CI step.
4. **Policy Closes the Loop**: In-toto attestations signed by role-scoped SVIDs allow policy engines like Conforma to verify *who was authorized to make each claim*.

Thank You!

Questions & Discussion

Andrew McNamara (amcnamar@redhat.com) • **Maia Iyer** (miyer@redhat.com)

Working Demo & Helm Charts

`arewm/slsa-konflux-example`

branch: kubecon-na-2026-your-cis-mistaken-identity

Presentation & Code

`arewm/presentations`

Explore the Repo: The Full Demonstration Arc

Everything we didn't have time to run live is in the demo repo:

- **Act 3 — Same-Namespace API Gating & Separation of Duties:** Zot OCI push gating (rogue task `403`, vetted builder `202`), secretless CVE database, OPA tests proving builder forgery is blocked.
- **Act 4 — The Managed Release Boundary:** Managed-tenant release pipeline, dual-gated release SVID, keyless VSA signing into Rekor, transparency-log verification.
- **The complete arc:** `./demo/run-demo.sh` runs Acts 0 through 4 end-to-end — pre-flight baseline, ambient breakdown, admission, API gating, and dual-gated release.
- **Also in the repo:** Helm charts for every policy (`charts/admission-policy/`, `charts/spiffe-spire/`), the Conforma separation-of-duties policy, and an idempotent `setup-demo.sh` for a KinD + Konflux + SPIRE + Sigstore cluster.
- **Going further:** the legacy-service token exchange and policy-gated release signing designs are in the repo, alongside everything else we showed today.

```
git clone https://github.com/arewm/slsa-konflux-example
git checkout kubecon-na-2026-your-cis-mistaken-identity
./demo/run-demo.sh              # full arc, or --act N for a single act
```

Scan to clone: github.com/arewm/slsa-konflux-example

branch `kubecon-na-2026-your-cis-mistaken-identity` · full setup guide in `demo/README.md`