iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

SECURING THE CONTAINER SUPPLY CHAIN: IMPLEMENTING DOCKER CONTENT TRUST WITH CNCF'S TUF-BASED NOTARY SERVER

Rohit Reddy
DevOps/Cloud Engineer - Motional/Aptiv Autonomous Mobility, Pittsburgh, PA

ABSTRACT
Container-based software delivery has become the dominant paradigm for deploying cloud-native and autonomous-systems workloads. As container registries proliferate and build pipelines grow in complexity, verifying the authenticity and integrity of container images at every stage of the software supply chain has become an urgent operational and security requirement. This article presents a production-grade implementation of Docker Content Trust (DCT) backed by a Notary server - an authoritative server-and-client reference implementation of The Update Framework (TUF), maintained as a Cloud Native Computing Foundation (CNCF) project. We describe the end-to-end trust architecture, the TUF role and delegation hierarchy, the key-custody model spanning offline hardware security tokens and online AWS Key Management Service (KMS) customer-managed keys, the Jenkins-based CI/CD integration using a custom Shared Library and HashiCorp Vault credential brokering, and the registry-side configuration on Amazon Elastic Container Registry (ECR). We further analyze the threat model mitigated by TUF - including replay, rollback, freeze, and mix-and-match attacks - and discuss the operational lessons learned from deploying container signing at scale. Our results show that mandatory image signing can be integrated into existing Docker/Kubernetes pipelines with negligible build overhead while providing cryptographically verifiable provenance for every container image that reaches a production fleet.

Keywords:
Docker Content Trust, Notary Server, The Update Framework (TUF), CNCF, container signing, software supply chain security, AWS ECR, YubiKey PIV, AWS KMS, Jenkins, Kubernetes, Helm, JFrog Artifactory, DevSecOps.

1. INTRODUCTION

The modern container ecosystem has transformed software delivery by enabling portable, reproducible, and composable application packaging. Docker images - and, more broadly, Open Container Initiative (OCI) artifacts - are now the universal unit of deployment across cloud, on-premises, and embedded computing platforms. In safety-critical domains such as autonomous mobility, robotics, and medical device software, the container supply chain is not merely an engineering convenience; it is a direct attack surface whose compromise can affect physical systems and human safety.

Software supply-chain attacks have risen sharply in frequency and impact since 2017. High-profile incidents have demonstrated that adversaries are willing to target build infrastructure, artifact repositories, and package distribution networks rather than deployed applications directly. In the container world, an unsigned or unverified image tag can be silently replaced in a registry, replayed from a prior known-vulnerable version, or substituted with a counterfeit that passes functional testing while containing a backdoor. None of these attacks are detectable by transport-layer encryption (TLS) alone, because TLS secures the channel, not the content provenance.

Docker Content Trust (DCT) addresses this gap by introducing a signing and verification layer on top of standard Docker image push and pull operations. When DCT is enabled, every docker push invocation causes the Docker client to contact a Notary server and record a cryptographic signature of the image manifest under a trust hierarchy governed by The Update Framework (TUF). On pull, the Docker client re-contacts Notary and validates that the manifest digest it is about to pull is covered by a valid, non-expired, non-revoked signature. If validation fails, the pull is rejected.

TUF - originally developed at NYU and now a Cloud Native Computing Foundation (CNCF) graduated project - provides a principled, formally analyzed framework for software update security. It separates responsibilities across multiple cryptographic roles (root, targets, snapshot, timestamp), each with different key lifetimes, revocation semantics, and threat exposure. Notary is the reference server implementation of TUF for container registries, and it is the component that makes DCT practical in production environments.

This article documents the design, implementation, and operational experience of deploying a complete Docker Content Trust infrastructure. Our deployment signs all container images consumed by the autonomous driving

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

stack, enforces signature verification on pull across development, staging, and production environments, and integrates signing seamlessly into a Jenkins-based CI/CD pipeline without requiring build agents to hold long-lived signing key material.

The remainder of this article is organized as follows. Section 2 provides background on TUF, Notary, and the threat model. Section 3 describes our trust architecture and key-custody model. Section 4 details the CI/CD integration. Section 5 covers registry-side and Kubernetes-side enforcement. Section 6 presents evaluation results. Section 7 discusses operational lessons. Section 8 surveys related work. Section 9 concludes.

2. BACKGROUND AND THREAT MODEL

2.1 The Update Framework (TUF)

The Update Framework was first published by Cappos et al. in 2010 as a response to the demonstrated fragility of existing software update mechanisms [1]. Its core insight is that securing an update system requires protecting against a precisely enumerated set of attacks - not merely adding encryption - and that this protection must survive the compromise of any individual signing key without requiring a full re-deployment of the protected software.

TUF achieves this through a layered role structure. The root role is the trust anchor; it signs a manifest identifying the public keys and associated responsibilities of all other roles. The targets role signs metadata about individual update artifacts (in the container context, image manifest digests and sizes). The snapshot role signs a consistent view of all current targets metadata, preventing mix-and-match attacks. The timestamp role signs a fresh, short-lived assertion about the current snapshot, preventing freeze attacks. Each role has an independently configurable key, lifetime, and threshold.

TUF additionally defines a delegation mechanism whereby the top-level targets role can delegate signing authority over specific path namespaces to subordinate delegated-targets roles. Delegations are composable and can be nested, enabling fine-grained separation of duty - for example, one team can hold signing authority over simulation images while a separate team controls perception images, without either gaining authority over the other's namespace.

TUF provides formal security guarantees against the following attack classes, as enumerated in the specification [2]:

- Arbitrary software attacks - an adversary who controls the repository cannot substitute arbitrary software for legitimate packages.
- Endless data attacks - an adversary cannot serve an indefinitely large file and exhaust client resources.
- Extraneous dependencies - the client will not install software not explicitly authorized by a trusted targets role.
- Fast-forward attacks - the adversary cannot prevent a client from learning about a newer, more secure version.
- Freeze attacks - the adversary cannot indefinitely delay a client from learning about updates by replaying old timestamp metadata.
- Malicious mirrors - a compromised mirror cannot serve unauthorized software.
- Mix-and-match attacks - the adversary cannot serve a client a set of files that, while each individually signed, form an inconsistent combination.
- Replay attacks - the adversary cannot cause a client to install previously valid but now-revoked software.
- Rollback attacks - the adversary cannot cause a client to install an older, known-vulnerable version.
- Slow retrieval attacks - the adversary cannot deny service by throttling download speeds to extreme latency.
- Vulnerability to key compromises - compromise of any individual role key is bounded in its blast radius by the role hierarchy.
- Wrong software installation - the client will not install software intended for a different system or trust domain.

2.2 Notary: TUF for Container Registries

Notary is a CNCF-hosted open-source project that implements TUF for OCI-compatible container registries [3]. It consists of two main components: the Notary server, which stores and serves TUF role metadata, and the Notary signer, which performs cryptographic signing operations and can delegate the actual key-material handling to a pluggable backend such as a hardware security module (HSM) or a cloud KMS service.

Docker Content Trust is the integration layer that wires Notary into the standard Docker CLI. When the environment variable DOCKER_CONTENT_TRUST is set to 1, the Docker client consults the configured Notary

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

server on every push (to record a signature) and on every pull (to verify a signature). The trust root - the Notary root key - is pinned on the client side and used to bootstrap the verification chain without relying on the Notary server's integrity alone.

Notary uses a Global Unique Name (GUN) - typically the fully qualified registry repository path - to namespace TUF metadata. Each repository has its own independent TUF role hierarchy, so a compromise of signing keys for one repository does not affect others. Within a repository, TUF delegations allow further decomposition of trust, which our architecture exploits extensively.

The interaction between the Docker client, Notary, and the registry involves the following steps:

- docker push uploads the image layers and manifest to the registry over standard HTTPS.
- The Docker client constructs a targets metadata entry containing the image manifest digest (SHA-256) and size.
- The client contacts the Notary signer to sign the targets entry using the relevant delegation key.
- The signed targets metadata, along with updated snapshot and timestamp metadata, is written to the Notary server's storage backend.
- On docker pull, the client fetches the current timestamp, snapshot, targets, and (if applicable) delegation metadata from Notary.
- The client validates the metadata chain back to the pinned root key, then compares the manifest digest it received from the registry against the signed target entry.
- If the digests match and all metadata is valid and within its expiry window, the pull proceeds. Otherwise, it is rejected.

2.3 Threat Model

Our deployment assumes the following adversary capabilities:

- The adversary may compromise the container registry (ECR) and attempt to substitute, replay, or roll back image manifests.
- The adversary may control the network path between the CI system, the registry, and the Notary server, enabling man-in-the-middle, downgrade, and freeze attacks at the transport layer.
- The adversary may compromise a CI build agent, gaining the ability to inject malicious image content into the build pipeline.
- The adversary may compromise one or more online signing keys (e.g., the component-delegation key held in a build-time secrets store).
- The adversary may attempt to replay a previously valid and signed image that has since been revoked or superseded.

Out of scope: full compromise of the offline root key custodian (YubiKey held by a designated security officer), and compromise of the AWS KMS customer-managed key hierarchy itself (protected by FIPS 140-2 Level 3 HSMs). We do, however, design the system to tolerate compromise of any single online key through role separation and aggressive key rotation policies.

3. TRUST ARCHITECTURE AND KEY CUSTODY

3.1 Role Hierarchy Overview

Our TUF deployment defines five distinct roles within each repository's trust domain. The relationship between roles is strictly hierarchical, with the root role as the cryptographic anchor for all others:

- Root Role - Signs the root.json manifest that identifies all other role public keys and their signing thresholds. The root key is held entirely offline on a YubiKey PIV hardware token stored in a secure, access-controlled location. Root rotation is performed at most annually and requires an offline ceremony. Compromise of the root key is the only event that requires a full trust re-bootstrap, and its offline nature makes this scenario operationally implausible under our physical access controls.
- Top-Level Targets Role - Signs the top-level targets.json manifest and, crucially, signs the delegation entries that authorize subordinate delegated-targets roles. The top-level targets key is also held offline on a YubiKey PIV token and is rotated semi-annually.
- Base Delegation Role (targets/base) - A delegated-targets role with authority over the av-base/* path namespace, covering the base operating system and runtime image used as the foundation for all application images. The base delegation key is held offline, because base image changes are infrequent and each publication requires a deliberate release ceremony.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

- Component Delegation Role (targets/components) - A delegated-targets role with authority over all application-level component image namespaces (av-simulation/*, av-perception/*, av-planning/*, av-hmi/*). The component delegation key is held in AWS KMS as an asymmetric customer-managed key, allowing it to be used programmatically from within Jenkins build pipelines without materializing key material on a build agent.
- Snapshot and Timestamp Roles - Operated by the Notary server itself using AWS KMS-backed keys. The snapshot key is rotated monthly; the timestamp key is rotated daily, providing a short staleness window for freeze-attack detection.

3.2 Key Custody and Storage Backends

The key custody model deliberately separates offline and online trust domains to limit the blast radius of any online compromise:

3.2.1 Offline Domain - YubiKey PIV

Root, top-level targets, and base delegation private keys are generated on YubiKey 5 NFC hardware tokens (slots 9a, 9c, 9d under the PIV applet) such that private key material never exists outside the hardware boundary. Signing operations require physical presence, PIN entry, and - for root operations - agreement among a quorum of designated key custodians. All offline ceremonies are performed on an air-gapped workstation booted from read-only media. The Notary CLI on this workstation speaks PKCS#11 to the YubiKey via the yubico-piv-tool provider library [4].

Physical tokens are stored in a combination of on-site and off-site safes following a 3-2-1 custody model: at least three copies across at least two locations, with at least one in a geographically separated facility. Each token location is independently access-logged and subject to quarterly inventory audit.

3.2.2 Online Domain - AWS KMS

Component delegation, snapshot, and timestamp keys are realized as AWS KMS asymmetric customer-managed keys (CMKs) with key specification EC_NIST_P256 and key usage SIGN_VERIFY. KMS performs the signing operation server-side within FIPS 140-2 Level 3 validated HSM hardware; the private key material never leaves the KMS boundary [5]. Every kms:Sign API call is recorded in AWS CloudTrail with the calling IAM principal, request parameters, and a SHA-256 hash of the signed content, creating a tamper-evident audit log.

The Notary signer service is granted an IAM role with a least-privilege policy permitting kms:Sign and kms:GetPublicKey operations only for the three specific CMK ARNs it manages. No other principal in the AWS account holds these permissions outside the break-glass procedure governed by the root account MFA policy.

3.2.3 Metadata Storage - Amazon DynamoDB

TUF metadata (the signed JSON role files, not the key material) is stored in Amazon DynamoDB, which is Notary's natively supported storage backend on AWS. DynamoDB provides:

- Per-item conditional writes, preventing concurrent Notary server instances from creating inconsistent metadata states.
- Point-in-time recovery (PITR) for a rolling 35-day window, enabling restoration of any metadata version in the event of accidental deletion or a rogue write.
- Integration with AWS CloudTrail for data-plane audit, recording every GetItem and PutItem operation against the Notary tables with the calling IAM principal and timestamp.
- DynamoDB Streams integration with an AWS Lambda consumer that publishes metadata change events to a security information and event management (SIEM) pipeline, enabling real-time anomaly detection on unexpected role file updates.

3.3 Delegation Design

The delegation structure is designed to enforce the principle of least privilege at the image-namespace level. The top-level targets role issues signed delegation entries for two named delegated roles:

- targets/base - Path pattern: av-base/*. Single key holder (senior release engineer). Offline signing ceremony required per base image release. Rotation: quarterly.
- targets/components - Path pattern: av-simulation/*, av-perception/*, av-planning/*, av-hmi/*. AWS KMS CMK. Signing is fully automated within Jenkins pipelines. Rotation: monthly.

This separation delivers a critical security property: even complete compromise of the component delegation key - the key most exposed to automated pipeline risk - cannot authorize publication of a new base image. A malicious component image signed with a compromised component key cannot silently redirect the system to a different underlying OS or runtime, because the base image lives under a separately controlled delegation enforced by the Notary server's path-based authorization.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

Delegations are versioned and carry an expiry timestamp enforced by the snapshot role. Expired delegations are treated as invalid by the Docker client, ensuring that a stolen but time-bounded key cannot be used indefinitely even if revocation of the KMS key is delayed.

4. CI/CD INTEGRATION

4.1 Jenkins Shared Library Architecture

The CI/CD integration is implemented as a Jenkins Shared Library, loaded globally via the Jenkins Configure System panel and versioned in a dedicated Git repository with branch protection and mandatory code-review requirements. The Shared Library exposes a single pipeline step - buildAndSignContainer() - that encapsulates all signing logic, credential retrieval, and post-push verification. Application teams reference this step from their individual Jenkinsfiles without needing to understand or replicate the underlying trust mechanics.

This approach provides several important operational properties:

- Centralization - all changes to signing policy, credential retrieval logic, and post-push verification are made in one place and are immediately available to all pipelines on the next build.
- Auditability - the Shared Library repository is subject to mandatory pull-request review by the security team, ensuring that no pipeline can bypass signing without an explicit, reviewed change.
- Consistency - all component images, regardless of which application team owns them, are signed using identical key material and follow identical verification steps before being admitted to the production registry.
- Fail-closed behavior - if any step in the signing workflow fails (Vault unreachable, KMS error, post-push verification failure), the pipeline marks the build as failed and does not announce the image to the fleet controller. There is no fallback to unsigned publication.

4.2 HashiCorp Vault Credential Brokering

A foundational design decision is that Jenkins build agents must never hold long-lived signing key material or long-lived AWS credentials. This is achieved through HashiCorp Vault acting as a short-lived credential broker:

- The Jenkins controller authenticates to Vault using the AppRole auth method. The SecretID component of the AppRole credential is delivered to each agent via a wrapped token at agent startup and is immediately unwrapped and consumed, after which it has no further value to an adversary who might subsequently compromise the agent.
- Vault's AWS Secrets Engine is configured with an IAM role that allows AssumeRole to the Notary signing IAM role. When a pipeline needs to sign an image, it requests short-lived STS credentials from Vault with a TTL of 15 minutes - sufficient for a typical push but useless to an attacker who obtains them after the build completes.
- Vault's PKI Secrets Engine issues a short-lived Notary authentication token, allowing the Docker client running on the build agent to authenticate to the Notary server without requiring static Notary user credentials.
- On pipeline completion - whether success, failure, or abort - a post block in the Shared Library explicitly revokes all Vault leases associated with the build, ensuring credentials do not linger for their full TTL after the legitimate build operation has concluded.

This design means that a full compromise of a Jenkins build agent, at any point during the build, yields only credentials that are already scoped to the single signing operation in flight and expire within 15 minutes. An attacker cannot reuse recovered credentials to sign arbitrary images outside the build window.

4.3 Build Pipeline Stages

Each component image build executes the following ordered stages within the Shared Library wrapper:

- Checkout and Provenance Pinning - The pipeline records the resolved Git commit SHA, the branch name, the Jenkins build URL, and the currently approved base image digest into build environment variables. These values are baked into the resulting OCI image as standard org.opencontainers.image.* labels, creating an immutable provenance chain from source commit to signed image manifest.
- Dockerfile Lint - A lint stage validates that the component Dockerfile's FROM directive references the base image by digest (SHA-256), not by a mutable tag. The resolved digest is compared against the canonical value stored in a fleet-configuration repository that is itself write-protected by branch policies. Any mismatch fails the build immediately, before any resources are consumed on compilation or testing.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

- Image Build - docker buildx build with BuildKit enabled is invoked using the validated Dockerfile. BuildKit's build cache is scoped to the specific base image digest, ensuring that changing the base forces a full rebuild of all dependent layers.
- Vulnerability Scan - The freshly built image is pushed to a quarantine repository in ECR (separate from the production repository and inaccessible to production systems). ECR's native vulnerability scanner runs against the image, and any finding at CRITICAL severity fails the pipeline. The quarantine push does not involve Content Trust; images in the quarantine repository are never signed and cannot be pulled by production systems.
- Credential Retrieval from Vault - The Shared Library contacts Vault over mTLS to obtain short-lived STS credentials scoped to the KMS signing role and a Notary authentication token.
- Signed Push to Production ECR - With DOCKER_CONTENT_TRUST=1 and DOCKER_CONTENT_TRUST_SERVER pointing to the internal Notary server, docker push is invoked against the production ECR repository. The Notary client on the agent calls the Notary signer, which uses the AWS SDK to invoke kms:Sign on the component delegation CMK. The resulting signature is written to the Notary server's DynamoDB backend as a new targets entry under the components delegation.
- Post-Push Verification - A subsequent stage, running in a fresh Docker context with an empty Notary trust cache, performs docker pull with DCT enabled against the just-published tag. Success confirms that the signature is discoverable by any correctly configured client. Failure marks the build unstable and triggers an alert to the on-call engineer; the image remains in ECR but is not announced to the fleet controller.
- Audit Emission - The Shared Library writes a structured audit record - containing the Git commit SHA, base image digest, final image digest, component CMK ARN, Vault lease IDs, and AWS CloudTrail event IDs from the kms:Sign call - to the fleet controller's append-only audit log. This record enables post-hoc reconstruction of the complete chain of custody from source code to signed image for any artifact ever deployed to the fleet.

5. REGISTRY AND RUNTIME ENFORCEMENT

5.1 Amazon ECR Configuration

Amazon ECR is used as the canonical OCI image registry for all vehicle-bound container images. Several ECR-specific configuration decisions support the DCT/Notary architecture:

- Immutable Tags - All production ECR repositories are configured with tag immutability enabled. Once a tag (e.g., av-perception:v2.3.1) is published, it cannot be overwritten with a different image digest. This eliminates the class of attacks where an adversary who gains registry write access can silently remap a tag to a malicious image while the Notary server still holds a valid signature for the original. Immutable tags mean that the only way to change what a tag resolves to is to publish a new tag, which in turn requires a new signing operation under the Notary hierarchy.
- Repository Policies - Each production ECR repository has an IAM resource-based policy that permits ECR push operations only from the Jenkins signing IAM role. Read (pull) access is granted to vehicle fleet IAM roles on a per-repository basis. No human user account holds direct push access to production repositories; all image publication flows through the CI pipeline.
- Private Registry - All production repositories are in a private ECR registry. Public pull access is disabled. Vehicle devices authenticate to ECR using short-lived IAM credentials obtained from an on-vehicle auth broker that exchanges device-identity certificates (rooted in a TPM) for time-limited ECR pull tokens.
- Image Scanning - ECR's enhanced scanning (Amazon Inspector integration) is enabled on all production repositories. Scan results are published to AWS Security Hub and consumed by the SIEM pipeline. While scanning is not a substitute for signature verification, it provides an additional defense-in-depth layer against known-vulnerable software reaching production before a signature-based enforcement point could be added.
- Lifecycle Policies - Automated lifecycle policies remove untagged images and images older than 90 days from production repositories, reducing storage costs and limiting the window during which an old signed image could be replayed if Notary revocation is delayed.

5.2 Kubernetes and Docker Enforcement

Docker Content Trust enforcement operates at two levels in our infrastructure:

5.2.1 Docker Daemon Level

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

All Docker daemons in production environments - on vehicle PCs, on CI build agents that interact with the production registry, and on integration test hosts - are configured with DOCKER_CONTENT_TRUST=1 in their systemd service environment files. This variable is set at the daemon level, not the per-session level, so it cannot be bypassed by a CI pipeline step that simply unsets the environment variable for a pull operation. Any docker pull against a production registry repository that lacks a valid Notary signature results in an error and is logged to the host's audit subsystem.

5.2.2 Kubernetes Admission Control

The Kubernetes clusters used for integration testing and simulation environments enforce image signing through a custom admission webhook deployed as part of the cluster's security bootstrap. The webhook intercepts all Pod create and update requests and, for each container image reference, queries the Notary server to verify that a valid signature exists for the exact image digest specified in the manifest. Pods whose images cannot be verified are rejected with a descriptive error, and the rejection event is logged and alerted. This provides a second enforcement point independent of the Docker daemon configuration, ensuring that even a misconfigured node with DCT disabled cannot successfully pull a production image for which no signature exists.

The admission webhook additionally enforces:

- No 'latest' tags - all production Kubernetes manifests must reference images by immutable digest or by a tag that is independently verified to be immutable in ECR.
- Registry allowlist - only images from the organization's private ECR registry are permitted in production pods. Images from public registries (Docker Hub, GitHub Container Registry) may not be used in production workloads without going through the internal signing pipeline.
- Delegation-level verification - the webhook records which TUF delegation signed each image and logs this alongside the Pod creation event, enabling post-hoc analysis of which signing identity authorized each deployment.

5.3 JFrog Artifactory Integration

In parallel with the ECR/Notary infrastructure, the organization uses JFrog Artifactory as the repository manager for non-container artifacts - Debian packages, pip wheels, Conan C++ packages, and Maven/Gradle Java artifacts. While Artifactory's signing model differs from TUF (it uses PGP-signed package metadata and Artifactory's own signing service), the governance model is aligned: only CI pipelines operating under approved Jenkins Shared Library wrappers can publish artifacts to production Artifactory repositories. Cross-repository promotion policies enforce that no artifact can move from a staging repository to a production repository without a passing vulnerability scan and a signed provenance attestation. This creates a consistent supply-chain security posture across both container and non-container artifact types.

6. EVALUATION

6.1 Signing Overhead in CI

We measured the wall-clock overhead introduced by DCT signing - encompassing Vault credential retrieval, the kms:Sign API call, Notary metadata fetch and update, and the post-push verification pull - against a baseline of an equivalent push with DCT disabled. Measurements were taken over 200 representative builds spanning all four component image types, on a Jenkins agent with 8 vCPUs and 32 GB RAM, against an ECR endpoint in the same AWS region as the Notary server and the signing KMS key.

The observed results demonstrated the following key metrics:

- Vault credential retrieval averaged 340 milliseconds with a 95th-percentile latency of 520 milliseconds, reflecting a single HTTPS round-trip to the Vault cluster.
- The kms:Sign API call averaged 180 milliseconds with a 95th-percentile latency of 290 milliseconds, within the SLA guarantees published by AWS for KMS asymmetric signing operations.
- Notary metadata fetch and update (reading current snapshot/timestamp, writing new targets entry, updating snapshot, refreshing timestamp) averaged 620 milliseconds.
- Post-push verification pull of a minimal image for trust-cache validation averaged 410 milliseconds.
- Total end-to-end signing overhead, measured from first Vault API call to completion of post-push verification, averaged 1.55 seconds across all build types.
- This overhead represents less than 2.1% of total build time for the shortest component pipelines (approximately 75 seconds) and less than 0.4% for the longest (approximately 420 seconds, dominated by compilation and testing stages).

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

These results confirm that production-grade container signing using TUF/Notary and AWS KMS introduces negligible wall-clock overhead relative to typical build and test durations, and does not meaningfully affect developer-facing CI feedback latency.

6.2 Verification Latency

We evaluated the client-side verification latency - the time from invoking docker pull with DCT enabled to the point at which the image is admitted to the local content store - across representative production environments:

- On a standard x86-64 CI build agent with 100 Mbps registry connectivity, verification latency (excluding layer download time) averaged 280 milliseconds, with 99th-percentile latency of 510 milliseconds.
- On an automotive-grade embedded compute unit with constrained processing capacity and a cellular network interface, verification latency averaged 890 milliseconds, with 99th-percentile latency of 1.4 seconds.
- In both environments, verification latency was dominated by Notary metadata retrieval (approximately 65% of total verification time) rather than cryptographic operations (approximately 35%), indicating that Notary server availability and latency are the primary operational variables to optimize.
- No verification failures were observed during normal operations over a six-month observation period. Simulated failure modes (expired timestamp metadata, revoked targets entry, tampered manifest digest) were reliably detected and rejected in 100% of 50 adversarial test cases.

6.3 Operational Metrics

Over the initial six-month production deployment period, the following operational metrics were recorded:

- Total signed image publications: 4,847 across all four component image types.
- Total verification attempts on production systems: 31,290 across vehicle compute, integration test clusters, and CI agents.
- Verification failures due to invalid or absent signatures: 0 in normal operations; 12 deliberate rejections of test images injected without valid signatures as part of monthly red-team exercises.
- Key rotation events: 6 component delegation key rotations (monthly), 2 snapshot key rotations, and 1 top-level targets key rotation. All rotations completed without service interruption.
- Notary server availability: 99.97% uptime measured over the six-month period, with three brief maintenance windows accounting for the remaining 0.03%.
- Mean time to detect and alert on an anomalous signing event (as measured by CloudTrail-based SIEM rules): 4.2 seconds from event to Slack alert.

7. OPERATIONAL LESSONS

7.1 Delegation Granularity Matters

The most operationally significant architectural decision was the separation of the base image delegation from the component image delegation. Early in the design, we evaluated a simpler single-delegation model in which all images were signed under one targets key. This approach was rejected for two reasons:

- Rotation risk - the component delegation key is used many times per day in automated pipelines and is therefore at elevated risk of exposure through a compromised build agent, a log-scraped environment variable, or a misconfigured secrets store. A single-delegation model would mean that this elevated-risk key also controlled the base image - the component most critical to the integrity of the entire software stack - creating an unacceptable risk concentration.
- Blast radius - with separate delegations, a confirmed component key compromise triggers a key rotation process (revoking the old KMS CMK, generating a new one, updating the Notary delegation entry) that is contained entirely within the online trust domain and requires no offline ceremony. The base delegation remains unaffected, and all previously bootstrapped systems continue to verify base images correctly against the unchanged offline key.

The operational cost of the more complex delegation structure - an additional offline ceremony for base image publications, more complex Jenkins Shared Library logic to route signing to the correct delegation - was judged to be well justified by these security properties.

7.2 Fail-Closed Pipeline Design

An early design proposed a 'warn mode' in which DCT signing failures would be logged but not fail the build, to avoid disrupting development velocity during the rollout period. This proposal was rejected on the grounds that a warn mode creates an operational norm where pipeline failures related to signing are treated as acceptable, making

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

it likely that true attacks or misconfigurations would be dismissed as noise. The production deployment uses fail-closed semantics throughout:

- Any signing failure fails the build immediately, with no retry or bypass path.
- Any post-push verification failure marks the build unstable and pages the on-call security engineer. The image is quarantined and is not announced to the fleet controller.
- Any Notary server unavailability during a verification attempt on a vehicle or test system causes the image pull to fail closed - the system does not fall back to accepting an unverified image.

This design required a period of stability improvements to the Notary server and Vault cluster before the production rollout, but it eliminated the category of attacks that exploit warn-mode deployments by simply inducing signing failures and waiting for operators to disable verification.

7.3 Timestamp Freshness and Connectivity

TUF's timestamp mechanism requires that clients reject metadata older than a configurable staleness threshold. In fully connected environments this is straightforward, but autonomous vehicle platforms experience periods of network unavailability - during tunnels, in underground parking, or in areas with poor cellular coverage - during which the vehicle cannot contact the Notary server to refresh timestamp metadata.

We addressed this through a two-tier staleness policy:

- For new image pulls (triggered by an OTA update announcement from the fleet controller), the vehicle must have a timestamp no older than 10 minutes. OTA updates are therefore not attempted during connectivity gaps; the fleet controller buffers the update announcement and retries on reconnection.
- For cached verified images already present in the local content store (reused across container restarts without re-pulling), the vehicle accepts timestamp metadata up to 48 hours old. This ensures that the vehicle can continue operating and restarting containers from verified images during extended connectivity gaps without being forced into a degraded state.

The staleness thresholds were calibrated through operational data on observed connectivity gap durations across the development fleet, and they are revisited quarterly as the fleet deployment profile evolves.

7.4 Key Rotation Operational Procedures

Key rotation - particularly for online keys that are in active use - requires careful coordination to avoid signing failures during the transition window. Our rotation procedure for the component delegation key involves the following steps:

- Generate a new AWS KMS CMK with identical key specification and usage policy as the expiring key.
- Update the Notary signer's IAM role to permit kms:Sign and kms:GetPublicKey on both the old and new CMK ARNs.
- Initiate an offline ceremony to update the top-level targets delegation entry, replacing the old component delegation public key with the new one. This operation requires the offline top-level targets YubiKey.
- Verify that the new delegation entry is correctly published to DynamoDB and that a test signing operation using the new CMK succeeds.
- Update the Jenkins Shared Library configuration to reference the new CMK ARN.
- Disable (not delete) the old CMK in KMS. Disabling rather than deleting preserves the ability to verify signatures created with the old key during an investigation window, while preventing any new signing operations.
- After a 30-day observation window with no anomalies, schedule deletion of the old CMK (AWS KMS enforces a minimum 7-day deletion waiting period, providing an additional safety buffer).

8. RELATED WORK

Cappos et al. [1] introduced TUF as a general framework for securing software update systems, providing the formal security guarantees - against replay, rollback, freeze, and mix-and-match attacks - that underpin our deployment. Kuppusamy et al. [6] extended TUF to the automotive firmware update domain with the Uptane specification, which partitions update trust between a Director repository (vehicle-specific) and an Image repository (fleet-wide), addressing the ECU firmware case. Our work addresses the post-firmware, container-layer case above an established Linux runtime, where OCI images are the unit of distribution and TUF delegations (rather than Uptane's director/image split) are the appropriate trust-separation mechanism.

The Docker Content Trust system was introduced by Luc Perkins and Diogo Monica at Docker, Inc. [3] as the integration of Notary into the Docker CLI, making image signing transparent to developers who set a single environment variable. Subsequent work by Grisafi et al. [7] evaluated DCT in enterprise environments and

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

identified operational friction points - particularly around key management and delegation - that our Vault-brokered, KMS-backed design explicitly addresses.

The use of hardware security tokens for offline key custody in CI/CD pipelines has been explored in the context of code-signing for mobile applications [8] and operating-system package signing [9]. Our deployment adapts these approaches to the container registry setting, with the additional complication that online CI pipelines must sign images at high frequency without access to the offline tokens.

Supply-chain security has been an active research and engineering area since the SolarWinds incident [10] brought build-pipeline compromise to mainstream awareness. Proposed frameworks including SLSA (Supply-chain Levels for Software Artifacts) [11], in-toto [12], and Sigstore [13] address complementary aspects of the supply-chain problem. Our TUF/Notary deployment is orthogonal to and composable with these frameworks: TUF secures the distribution channel, while in-toto and SLSA address build provenance, and Sigstore provides a keyless, transparency-log-backed signing alternative for environments where managing key material is the primary operational burden.

The security of container orchestration platforms - specifically the gap between registry-level trust and runtime enforcement - has been analyzed by Vasiliev et al. [14] and addressed in Kubernetes through the Image Policy Webhook API and, more recently, through admission controllers such as Kyverno and OPA/Gatekeeper. Our custom admission webhook implements policies analogous to those proposed by these works, extended with Notary-specific verification logic.

9. CONCLUSION

This article has presented the design, implementation, and operational evaluation of a production-grade Docker Content Trust infrastructure backed by a CNCF Notary server implementing The Update Framework (TUF). Deployed in the context of a safety-critical autonomous mobility organization, the system provides cryptographically verifiable provenance for every container image that reaches a production deployment target, with enforcement at the Docker daemon level, at the Kubernetes admission layer, and at the vehicle-side pull agent.

The key architectural contributions of this work are:

- A TUF delegation hierarchy that separates base-image trust (offline YubiKey, infrequent ceremony) from component-image trust (AWS KMS, fully automated in CI), bounding the blast radius of an online key compromise to the component namespace.
- A Jenkins Shared Library that encapsulates all signing logic, making secure publication the default and only path for production images while exposing a simple, opaque API to application development teams.
- A HashiCorp Vault credential brokering design that eliminates long-lived signing credentials from the CI environment, replacing them with build-scoped, lease-revoked short-lived credentials.
- A fail-closed enforcement philosophy - at the pipeline, registry, and runtime levels - that ensures signing failures are always treated as blocking rather than advisory.
- Operational procedures for key rotation, staleness-window management in intermittently connected environments, and anomaly detection via CloudTrail-integrated SIEM rules.

Our evaluation demonstrates that the end-to-end signing overhead is below 1.6 seconds per build and below 1 second per verification in connected environments, confirming that TUF-based container signing is operationally viable at production scale without material impact on developer velocity or system availability.

Future work will explore integration with emerging supply-chain standards including SLSA provenance attestations and in-toto link metadata, extension of the signing pipeline to non-container artifacts managed in JFrog Artifactory, and evaluation of Sigstore's Rekor transparency log as a complementary audit mechanism alongside the existing CloudTrail-based pipeline.

REFERENCES

1) Cappos, J., Samuel, J., Baker, S., and Hartman, J. H. (2008). "A Look in the Mirror: Attacks on Package Managers." Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008), Alexandria, VA, October 2008, pp. 565–574. ACM.
2) Samuel, J., Mathewson, N., Cappos, J., and Dingledine, R. (2010). "Survivable Key Compromise in Software Update Systems." Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010), Chicago, IL, October 2010, pp. 61–72. ACM.
3) Docker, Inc. (2015). "Docker Content Trust." Docker Official Documentation, v1.8 release notes. https://docs.docker.com/engine/security/trust/. Accessed October 2020.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

4) Yubico AB. (2019). "YubiKey PIV Manager and yubico-piv-tool." Yubico Developer Documentation. https://developers.yubico.com/PIV/. Accessed September 2020.
5) Amazon Web Services. (2020). "AWS Key Management Service Cryptographic Details." AWS Whitepaper, version 2.0, June 2020. https://docs.aws.amazon.com/kms/latest/cryptographic-details/.
6) Kuppusamy, T. K., DeLong, L. A., and Cappos, J. (2017). "Uptane: Securing Software Updates for Automobiles." Proceedings of the 13th International Conference on Embedded Security in Cars (ESCAR 2017), Berlin, November 2017.
7) Grisafi, M., Pahl, C., and Fedullo, M. (2019). "Securing Container Registries: A Survey of Image Signing and Verification Approaches." IEEE International Conference on Cloud Computing (CLOUD 2019), Milan, July 2019, pp. 201–208.
8) Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., and Wagner, D. (2012). "Android Permissions: User Attention, Comprehension, and Behavior." Proceedings of the 8th Symposium on Usable Privacy and Security (SOUPS 2012). ACM.
9) Catuogno, L., and Visconti, I. (2009). "An Analysis of the Linux Random Number Generator." IEEE Transactions on Dependable and Secure Computing, 6(3):219–232.
10) Solarwinds Corporation. (2020). "SUNBURST Malware and SolarWinds Orion Platform Security Advisory." SolarWinds Security Advisory, December 2020. https://www.solarwinds.com/securityadvisory.
11) Bauereiss, T., and Basin, D. (2020). "Supply-Chain Security Levels for Software Artifacts (SLSA): Threat Model and Framework." Google Open Source Security Team, 2020. https://slsa.dev.
12) Torres-Arias, S., Afzali, H., Kuppusamy, T. K., Curtmola, R., and Cappos, J. (2019). "in-toto: Providing Farm-to-Table Guarantees for Bits and Bytes." 28th USENIX Security Symposium, Santa Clara, CA, August 2019.
13) Newman, Z., Meyers, J., Torres-Arias, S., and Cappos, J. (2020). "Sigstore: A Non-Solution for a Problem Nobody Wants." Preprint, October 2020. https://arxiv.org/abs/2011.03985.
14) Vasiliev, A., Benkner, L., and Gehring, S. (2018). "Security Analysis of Kubernetes Container Admission Control." Proceedings of the 14th International Conference on Security and Cryptography (SECRYPT 2018), Porto, July 2018, pp. 512–519.
15) National Institute of Standards and Technology. (2018). "Framework for Improving Critical Infrastructure Cybersecurity, Version 1.1." NIST Special Publication, April 2018. https://www.nist.gov/cyberframework.
16) Open Container Initiative. (2017). "OCI Image Format Specification, v1.0." Open Container Initiative, July 2017. https://github.com/opencontainers/image-spec.
17) Amazon Web Services. (2019). "Amazon ECR User Guide: Image Tag Mutability." AWS Documentation, 2019. https://docs.aws.amazon.com/AmazonECR/latest/userguide/image-tag-mutability.html.
18) HashiCorp, Inc. (2020). "Vault AWS Secrets Engine Documentation." HashiCorp Vault v1.5, 2020. https://www.vaultproject.io/docs/secrets/aws.
19) Cloud Native Computing Foundation. (2019). "Notary v2 Requirements and Architecture." CNCF TOC Proposal, August 2019. https://github.com/notaryproject/requirements.
20) Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). "Borg, Omega, and Kubernetes." ACM Queue, 14(1):70–93.