From Pipelines to Provenance: Reproducible Builds with Tekton

Divyanshu Agrawal
Red Hat

Shubham Bhardwaj
Red Hat

Open Source Summit India • Mumbai 2026

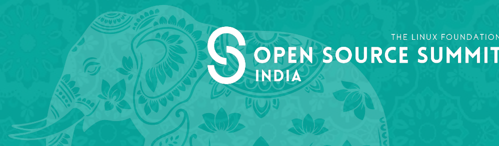
Reproducible Builds

Same source + Same build process = Identical artifact

Anyone can independently verify. No trust required.

If you can **reproduce** the build, you can **detect** the tampering.

The Evolution of Supply Chain Attacks



The target has shifted from the application to the build cycle.

Malicious code injected during creation compromises every downstream consumer automatically.

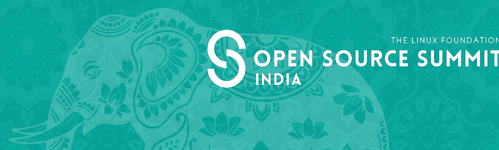
SolarWinds (2020)

- **Mechanism:** Injected a backdoor (SUNBURST) into the Orion Platform build system.
- **Impact:** The update was digitally signed by SolarWinds, making it appear legitimate to ~18,000 customers.
- **Lesson:** Trusting a vendor's signature isn't enough if their build environment is compromised.

xz utils (2024)

- **Mechanism:** A multi-year social engineering campaign led to a backdoor in the build scripts.
- **Impact:** Targeted SSH login mechanisms, potentially allowing unauthorized remote access at a massive scale.
- **Lesson:** Build system manipulation can be subtle and reside in obfuscated test files for years.

Why Reproducible Builds?



Certainty that software is genuine and has not been tampered with.

Security & Trust

Third parties can verify software hasn't been altered — increasing safety and reliability

Transparency in Development

Developers' code always works the same way — more consistent and trustworthy software

Protection of Build Infrastructure

Detect unauthorized changes to the build process early — before they reach users

Regulatory Compliance

Prove that binaries match source code — simplifying license and industry standard compliance

Resilience Against Attacks

Third-party verification prevents projects from being silently compromised

— reproducible-builds.org

The challenge: What actually breaks reproducibility?

1. Timestamps

Build clock baked into image config — different every run

2. Filesystem Paths

Embeds /home/alice/src into binaries

3. Build IDs

Generates a unique build ID every invocation

4. Base Image Drift

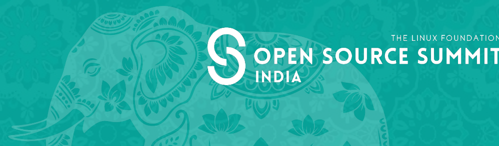
FROM ubuntu:latest changes every week

5. Source Drift

Building from branch HEAD vs pinned commit SHA

"Any single one is enough to break"

Achieving Reproducible Builds



Initial State

Triggering two pipeline runs results in **different images** due to non-deterministic build metadata.

The Modification

Pass required ***parameters*** to the PipelineRun to strip timestamps and stabilize the build environment.

Producer Compliance

The **Same Image** (identical digest) is produced every single time from the same commit.

Consumer Trust

SLSA Provenance generated automatically, providing verifiable evidence of the build's integrity.

"Deterministic builds turn mystery into math."

What are we building using Tekton?

PipelineRun: reproducible-build

Task Run: git-clone

Step: clone
Clones repo at pinned commit SHA

Workspace: shared-workspace
Results: commit, url
Params: url, revision

runAfter

Task Run: buildah

Step: build and Push
Builds image with reproducibility flags

Workspace: shared-workspace
Params: IMAGE, DOCKERFILE, BUILD_EXTRA_ARGS

binds

Shared Workspace
PVC (1Gi ReadWriteOnce)

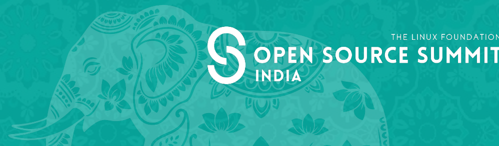
watches

Tekton Chains
Watches → Signs → SLSA Provenance
(automatic, zero pipeline changes)

emits

Pipeline Results
- IMAGE_URL
- IMAGE_DIGEST
- CHAINS-GIT_URL
- CHAINS-GIT_COMMIT

Let's build something.

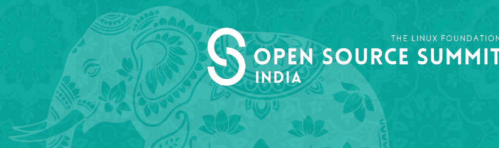


Same code. Same commit.

Two independent builds.

[Switch to terminal — start builds]

Same Code. Different Images.



Build #1

`sha256:59023a...`

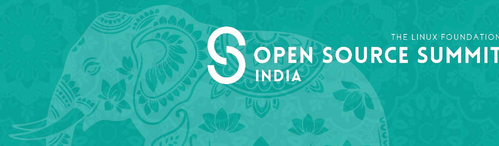
Build #2

`sha256:a7f31b...`

Buildah embedded real timestamps — different config, different digest.

Why?

The Fix: Dockerfile



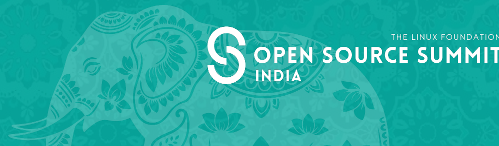
Dockerfile: Addressing challenges #2, #3, and #4

```
# Pin builder image by digest — challenge #4
FROM golang:1.22@sha256:1cf6c45b... AS builder
WORKDIR /src
COPY go.mod main.go ./

# challenge #2: -trimpath strips paths | challenge #3: -buildid= strips build ID
RUN CGO_ENABLED=0 go build -trimpath \
-ldflags="-buildid= -s -w" -o /app .

# Pin runtime image by digest — challenge #4
FROM cgr.dev/chainguard/static@sha256:67a1b00e...
COPY --from=builder /app /server
ENTRYPOINT ["/server"]
```

```
buildah --source-date-epoch 0 --rewrite-timestamp
```

Kills challenge #1 (Timestamps) to ensure metadata determinism.

Let's fix it.

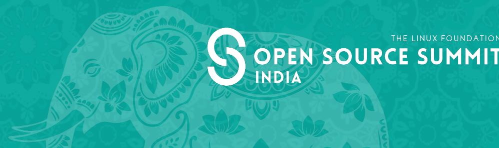


Dockerfile modifications.

+ two buildah flags.

[Switch to terminal — start reproducible builds]

Same Code. Same Image.



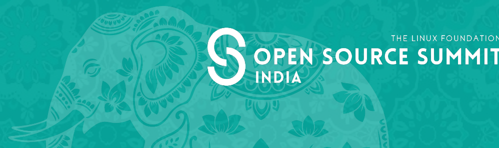
Build #1: `sha256:75d4b60a...`

Build #2: `sha256:75d4b60a...`

Before (naive)	After (+ two flags)
different digests	**identical digests**

Byte-identical.

Hermetic Execution



Self contained build, **Complete network isolation.**

```
apiVersion: tekton.dev/v1
kind: TaskRun
metadata:
annotations:
experimental.tekton.dev/execution-mode:
hermetic
spec:
taskRef:
name: verify-hermetic
```

Network access blocked

No surprise downloads

No data exfiltration

bit.ly/repro-builds

tekton-hermetic-builds

Consumer trust with SLSA Provenance

SLSA Provenance is a verifiable attestation that details where, when, and how a software artifact was produced

[SLSA provenance]

Tekton Chains

Automatic SLSA provenance — zero extra CI steps

1. Watches

Completed PipelineRuns

2. Reads

Type-hinted results (IMAGE_DIGEST, etc.)

3. Generates

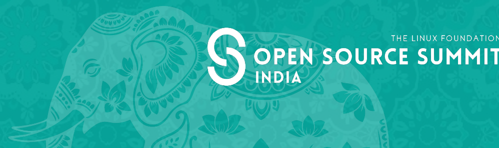
SLSA v1 provenance attestation

4. Signs

With cosign — stored as annotations

▸ ***You don't add provenance to your pipeline. Chains does it for you.***

SLSA Provenance: What Gets Recorded



Without flags

```
externalParameters:
   DOCKERFILE: ./Dockerfile
   CONTEXT: demo-app
   STORAGE_DRIVER: vfs
   taskRef: buildah
   BUILD_EXTRA_ARGS: ""
subject.digest:
sha256:546e122e...
```

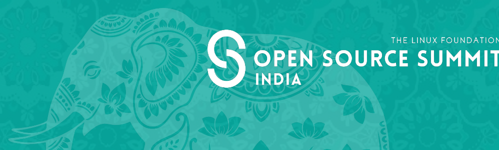
With flags

```
externalParameters:
   DOCKERFILE: ./Dockerfile
   CONTEXT: demo-app
   STORAGE_DRIVER: vfs
   taskRef: buildah
   BUILD_EXTRA_ARGS:  "--source-date-epoch 0
   --rewrite-timestamp"
subject.digest:
sha256:e59e9c35...
```

internalParameters (same for both builds)

`enableAPIFields: alpha, enableProvenanceInStatus: true, enforceNonfalsifiability: none`

Verification & Policy




```
$ cosign verify-blob-attestation \
--key k8s://tekton-chains/signing-secrets \
--type slsaprovenance1 \
--signature dsse-envelope.json \
/dev/null

Verified OK
```

Policy gate:

✓ Succeeded — ✓ Signed — ✓ Provenance valid — ✓ Signature verified

ALLOW

Beyond Tekton

These are universal build-tool techniques, not Tekton-specific:

`--source-date-epoch, -trimpath, pinned digests work everywhere`

What Tekton uniquely adds:

Hermetic Execution

A first-class primitive for complete isolation.

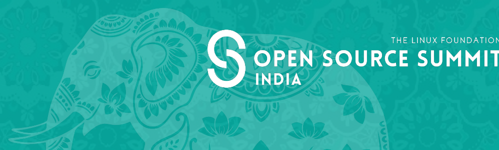
Structured Results

Automatic provenance generation via Chains.

SLSA Level 2

Compliance ready right out of the box.

Takeaways



01. Pin your base images by **digest** — 5 minutes

02. Add `--source-date-epoch 0 --rewrite-timestamp` to buildah

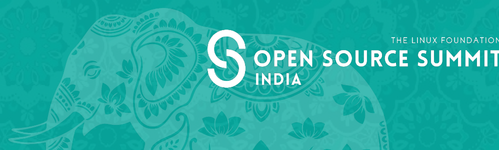
03. Strip build IDs and paths `(-buildid= -trimpath)`

04. Pin source by **commit SHA**, not branch

05. Add **Tekton Chains** for automatic provenance

Each step is incremental. Start with #1.

Resources



reproducible-builds.org
The reproducible builds project

slsa.dev
SLSA framework & levels

tekton.dev/docs/chains
Tekton Chains docs

Red Hat: Reproducible Builds
RHEL 10 documentation

go.dev/blog/rebuild
Perfectly reproducible Go builds

Thank You!

Questions?

Shubham Bhardwaj
github.com/infernus01

Divyanshu Agrawal
github.com/divyansh42

github.com/cloud-talks/reproducible-builds

Come find us! — **Red Hat** booth at KubeCon