Practical
Software Supply Chain Security

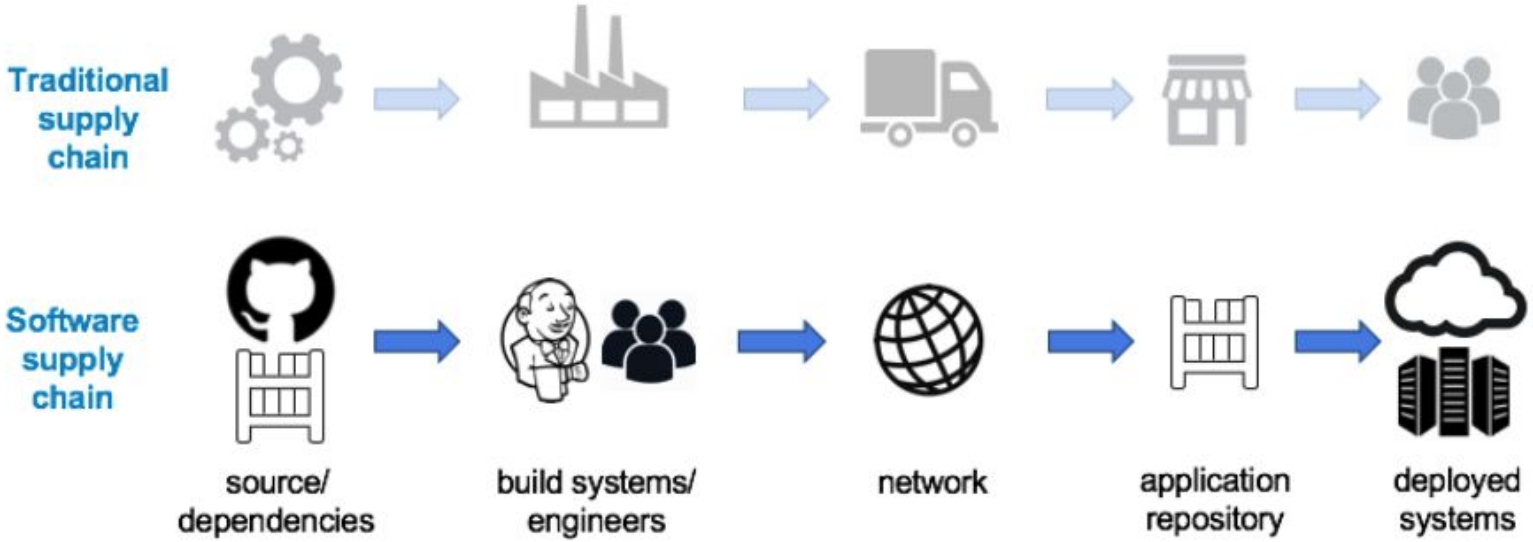
Mike Vainio

Family stats:

2 kids
4 cats
1 wife

Current State of Software Supply Chain Security

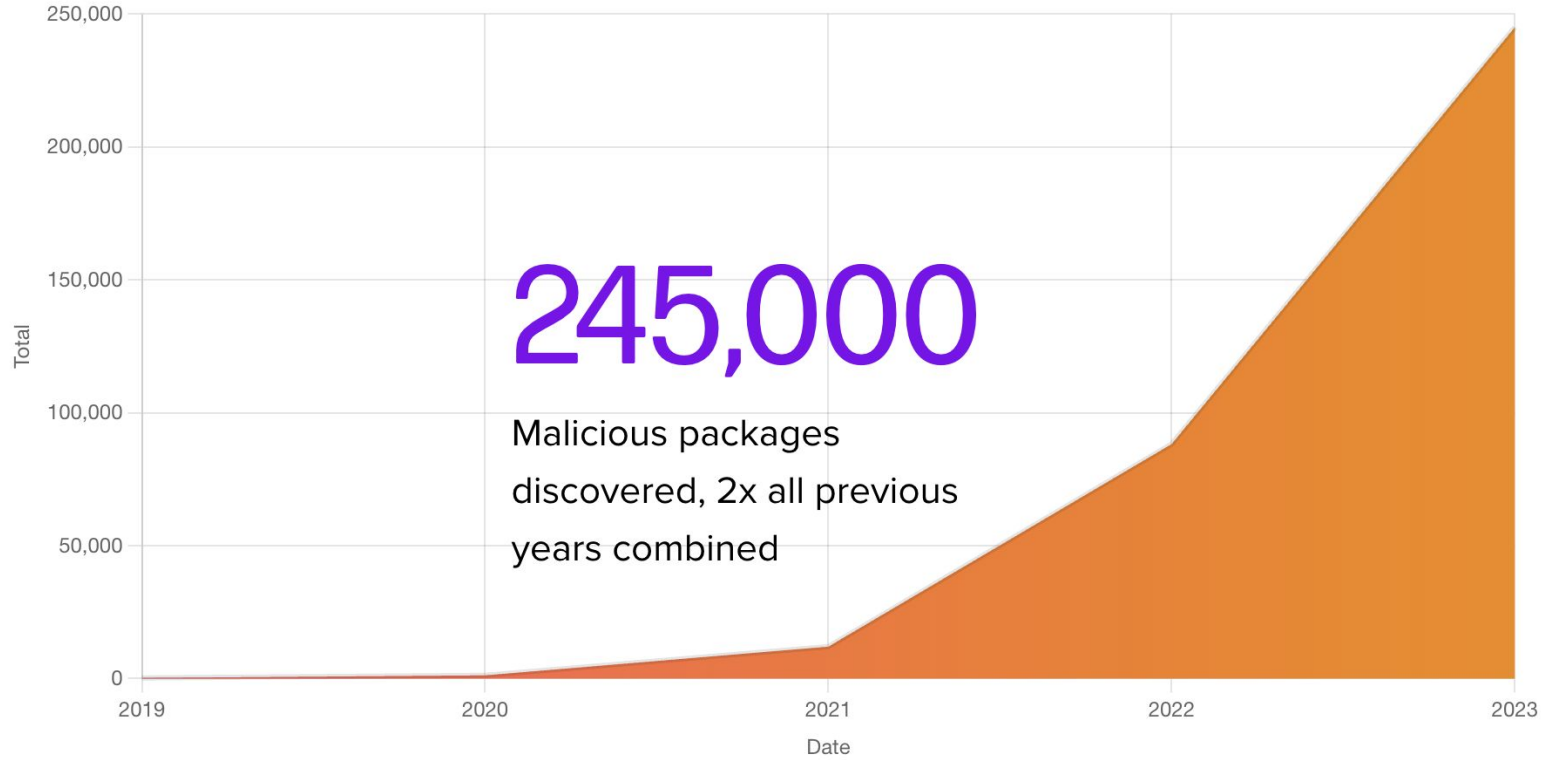
Software Supply Chain



Source: https://github.com/cncf/tag-security/blob/main/supply-chain-security/supply-chain-security-paper/CNCF_SSCP_v1.pdf

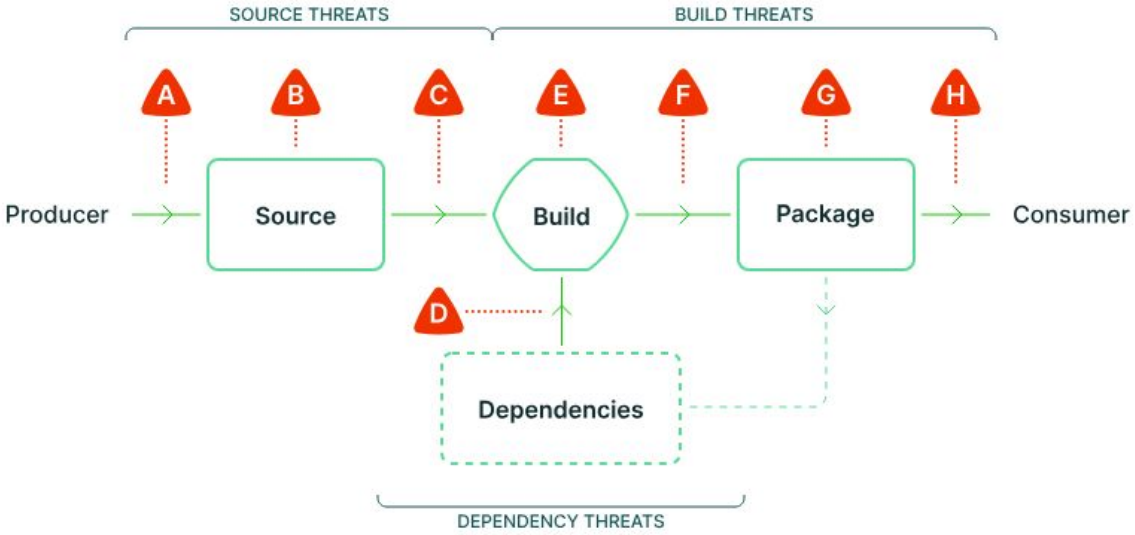
Supply chain attacks are becoming more common

FIGURE 1.7. NEXT GENERATION SOFTWARE SUPPLY CHAIN ATTACKS (2019-2023)



Source: https://www.sonatype.com/state-of-the-software-supply-chain/open-source-supply-and-demand

Threats all over the place



SOURCE THREATS

- A Submit unauthorized change
- B Compromise source repo
- C Build from modified source

DEPENDENCY THREATS

- D Use compromised dependency

BUILD THREATS

- E Compromise build process
- F Upload modified package
- G Compromise package registry
- H Use compromised package

Source: https://slsa.dev/spec/v1.0/threats-overview

Threats #2

	Integrity threat	Known example	How SLSA can help
A	Submit unauthorized change (to source repo)	**SushiSwap**: Contractor with repository access pushed a malicious commit redirecting cryptocurrency to themself.	Two-person review could have caught the unauthorized change.
B	Compromise source repo	**PHP**: Attacker compromised PHP's self-hosted git server and injected two malicious commits.	A better-protected source code platform would have been a much harder target for the attackers.
C	Build from modified source (not matching source repo)	**Webmin**: Attacker modified the build infrastructure to use source files not matching source control.	A SLSA-compliant build server would have produced provenance identifying the actual sources used, allowing consumers to detect such tampering.
D	Use compromised dependency (i.e. A-H, recursively)	**event-stream**: Attacker added an innocuous dependency and then later updated the dependency to add malicious behavior. The update did not match the code submitted to GitHub (i.e. attack F).	Applying SLSA recursively to all dependencies would have prevented this particular vector, because the provenance would have indicated that it either wasn't built from a proper builder or that the source did not come from GitHub.
E	Compromise build process	**SolarWinds**: Attacker compromised the build platform and installed an implant that injected malicious behavior during each build.	Higher SLSA levels require **stronger security controls for the build platform**, making it more difficult to compromise and gain persistence.
F	Upload modified package (not matching build process)	**CodeCov**: Attacker used leaked credentials to upload a malicious artifact to a GCS bucket, from which users download directly.	Provenance of the artifact in the GCS bucket would have shown that the artifact was not built in the expected manner from the expected source repo.

...

Source: https://slsa.dev/spec/v1.0/threats-overview

Software Bill of Materials

SBOM can help us understand what goes into an artifact

What's an SBOM?

```
[mike@linukka ~]$ syft nginx:alpine
 ✔ Loaded image
 ✔ Parsed image
 ✔ Cataloged packages              [66 packages]
NAME                    VERSION           TYPE
alpine-baselayout       3.4.3-r1          apk
alpine-baselayout-data  3.4.3-r1          apk
alpine-keys             2.4-r1            apk
aom-libs                3.6.1-r0          apk
apk-tools               2.14.0-r2         apk
brotli-libs             1.0.9-r14         apk
busybox                 1.36.1-r2         apk
busybox-binsh           1.36.1-r2         apk
ca-certificates         20230506-r0       apk
```

```
[mike@linukka verinotes]$ syft packages file:go.mod
 ✔ Indexed file system
 ✔ Cataloged packages              [29 packages]
NAME                                VERSION                                    TYPE
ariga.io/atlas                      v0.9.1-0.20230119145809-92243f7c55cb       go-module
entgo.io/ent                        v0.11.8                                    go-module
github.com/agext/levenshtein        v1.2.1                                     go-module
github.com/apparentlymart/go-textseg/v13  v13.0.0                              go-module
github.com/davecgh/go-spew          v1.1.1                                     go-module
github.com/go-chi/chi/v5            v5.0.8                                     go-module
github.com/go-chi/httplog           v0.2.5                                     go-module
```


```
{
  "spdxVersion": "SPDX-2.3",
  "dataLicense": "CC0-1.0",
  "SPDXID": "SPDXRef-DOCUMENT",
  "name": "go.mod",
  "documentNamespace": "https://anchore.com/syft/file/go.mod-f518a5fa-36fc-4f90-996d-ec1a9e3b7c14",
  "creationInfo": {
    "licenseListVersion": "3.21",
    "creators": [
      "Organization: Anchore, Inc",
      "Tool: syft-0.86.1"
    ],
    "created": "2023-11-20T12:39:12Z"
  },
  "packages": [
    {
      "name": "ariga.io/atlas",
      "SPDXID": "SPDXRef-Package-go-module-ariga.io-atlas-4ba436b91a46da78",
      "versionInfo": "v0.9.1-0.20230119145809-92243f7c55cb",
      "downloadLocation": "NOASSERTION",
      "filesAnalyzed": false,
      "sourceInfo": "acquired package info from go module information: /go.mod",
      "licenseConcluded": "NOASSERTION",
      "licenseDeclared": "NOASSERTION",
      "copyrightText": "NOASSERTION",
      "externalRefs": [
        {
          "referenceCategory": "PACKAGE-MANAGER",
          "referenceType": "purl",
          "referenceLocator": "pkg:golang/ariga.io/atlas@v0.9.1-0.20230119145809-92243f7c55cb"
        }
      ]
    }
```

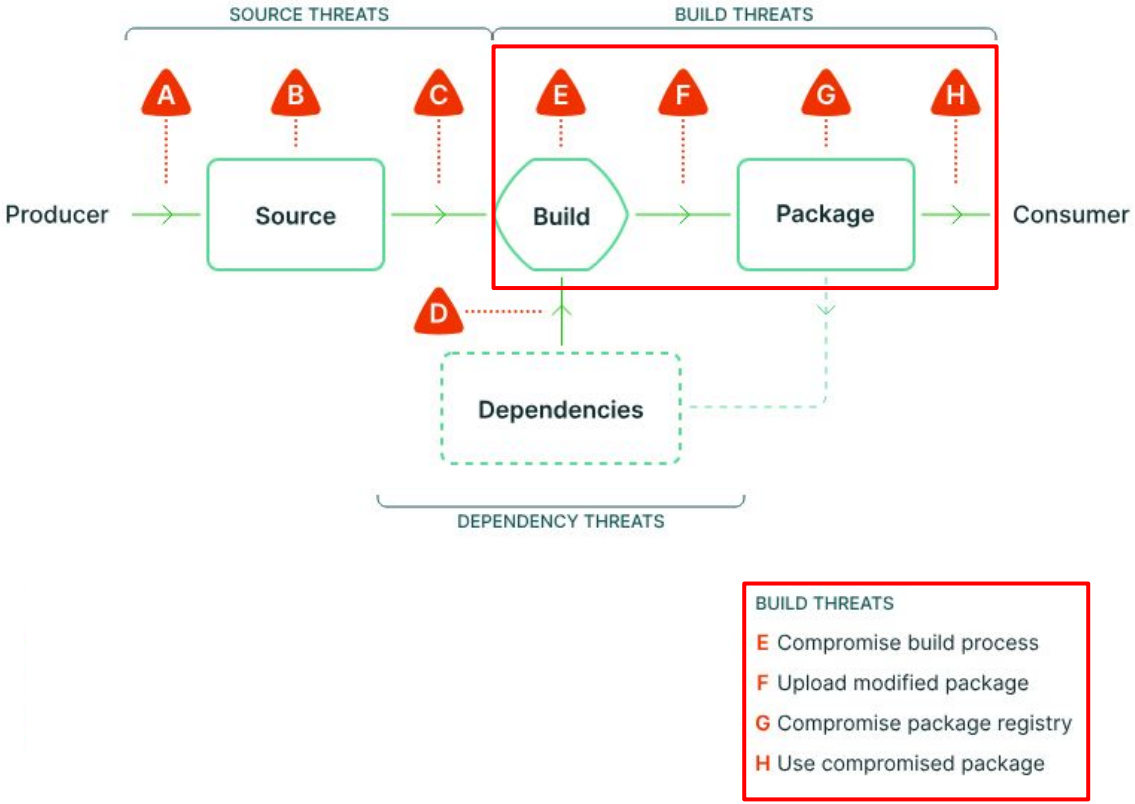
Supply-chain Levels for Software Artifacts

SLSA is a framework for software supply chain security

SLSA is not just about providing this metadata, it’s also about treating your build system as a production system

SLSA helps to trace an artifact back to it’s source

SLSA v1.0 – Build Track



Source: https://slsa.dev/spec/v1.0/threats-overview

SLSA Security Levels (for Build track)

Track/Level	Requirements	Focus
Build L0	(none)	(n/a)
Build L1	Provenance showing how the package was built	Mistakes, documentation
Build L2	Signed provenance, generated by a hosted build platform	Tampering after the build
Build L3	Hardened build platform	Tampering during the build

SLSA Provenance

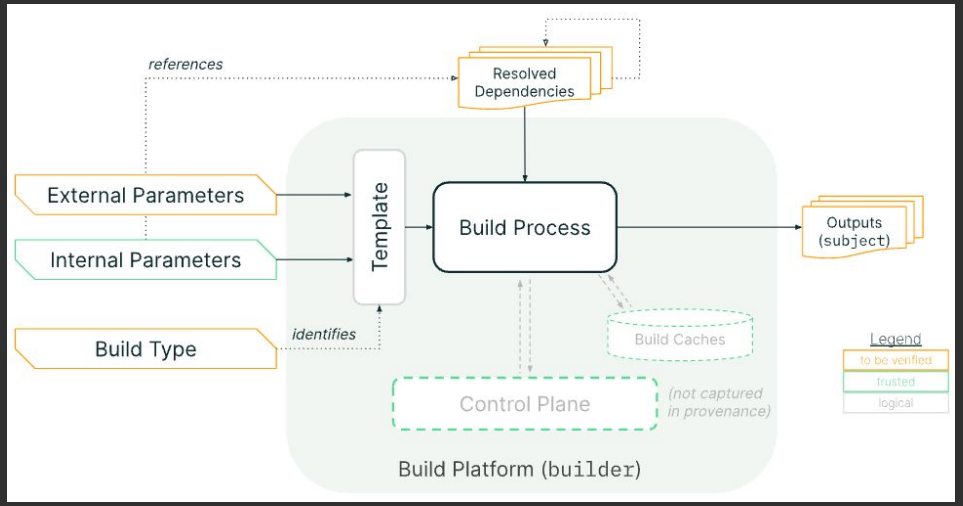
Provenance according to Google search:

the place of origin or earliest known history of something

SLSA provenance (according to SLSA.dev):

verifiable information about software artifacts describing where, when and how something was produced

SLSA Provenance



Source: https://slsa.dev/spec/v1.0/provenance


```
{
  "builder": {
    "id": "https://github.com/slsa-framework/slsa-github-generator/.github/workflows/
  },
  "buildType": "https://github.com/slsa-framework/slsa-github-generator/container@v1"
  "invocation": {
    "configSource": {
      "uri": "git+https://github.com/verifa/verinotes@refs/tags/v0.2.0",
      "digest": {
        "sha1": "d0a495e16d32c2bb125bf1481f747e711714199b"
      },
      "entryPoint": ".github/workflows/release.yaml"
    },
    "parameters": {},
    "environment": {
      "github_ref": "refs/tags/v0.2.0",
      "github_ref_type": "tag",
      "github_repository_id": "603037671",
      "github_repository_owner": "verifa",
      "github_sha1": "d0a495e16d32c2bb125bf1481f747e711714199b"
    }
  },
  "materials": [
    {
      "uri": "git+https://github.com/verifa/verinotes@refs/tags/v0.2.0",
      "digest": {
        "sha1": "d0a495e16d32c2bb125bf1481f747e711714199b"
      }
    }
  ]
}
```

How many here generate SBOMs?

How many here generate SLSA provenance?

Source: https://www.sonatype.com/state-of-the-software-supply-chain/software-supply-chain-maturity

75%

of leaders reported generating SBOMs for their applications

25%

of engineering professionals reported generating SBOMs for their applications

Sigstore

The Sigstore framework and tooling empowers software developers and consumers to securely sign and verify software artifacts

The project is backed by the Open Source Security Foundation (OpenSSF) under the Linux Foundation

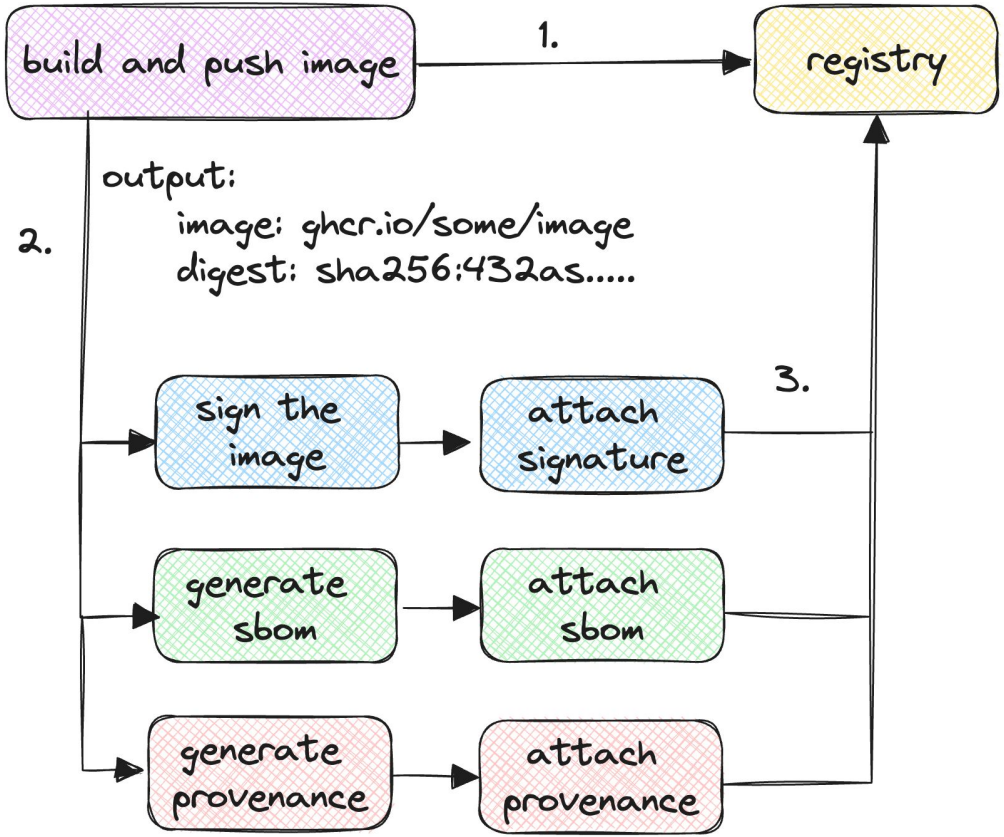
Main components:

- **cosign** CLI
- **Rekor** transparency log
- **Fulcio** code signing Certificate Authority.

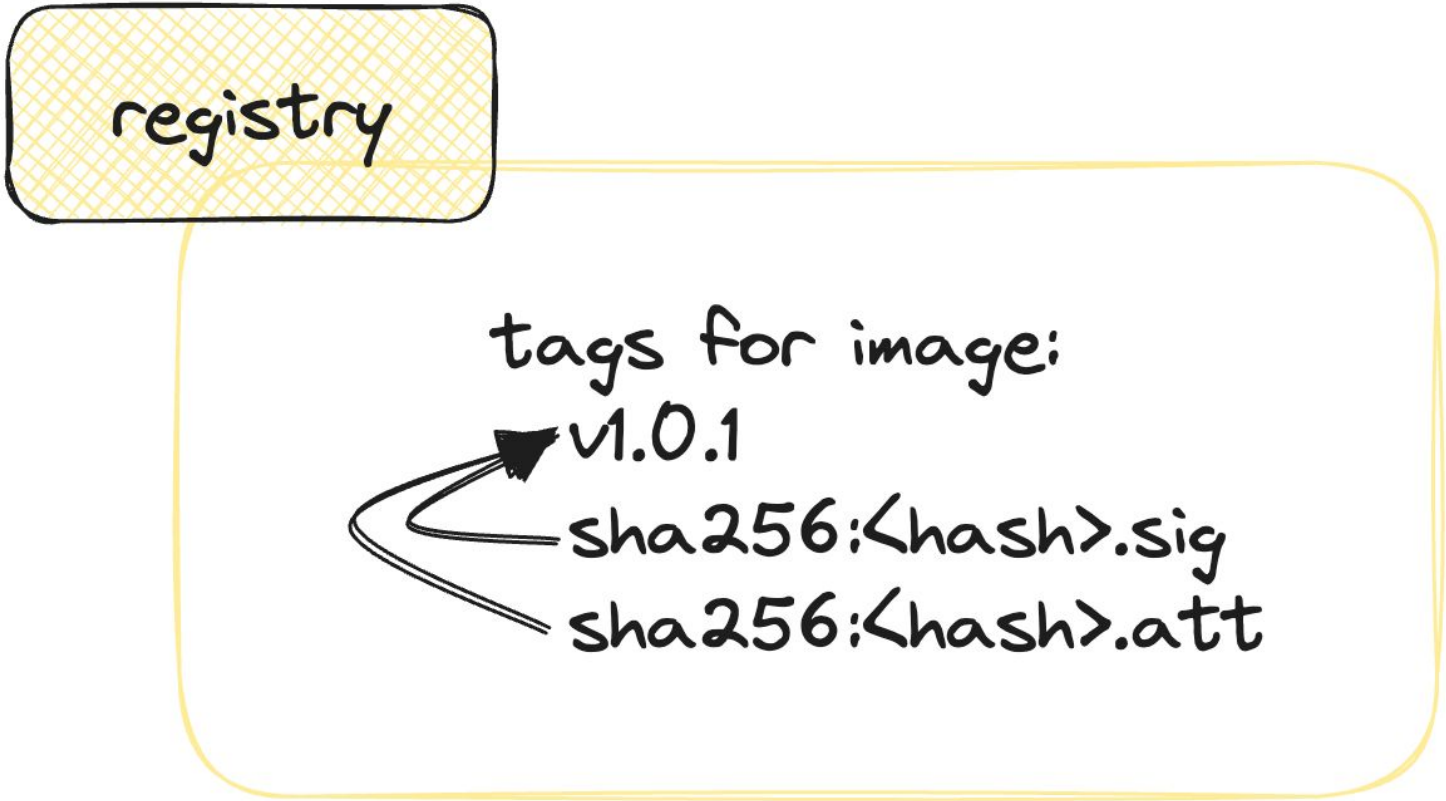
Source: https://www.sigstore.dev/how-it-works

Let’s build!

Pipeline Overview



Inside the Registry



Pipeline - Build and Push

```yaml
name: Release workflow
on:
  push:
    tags:
    - "*"

env:
  KO_DOCKER_REPO: "ghcr.io/${{ github.repository}}"

jobs:
  build:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write
    outputs:
      image: ${{ steps.build.outputs.image }}
      digest: ${{ steps.build.outputs.digest }}
    steps:
    - name: checkout repo
    - uses: actions/setup-go@fac708d6674e30b6ba41289acaab6d4b75aa0753 #v4.0.1
    - uses: ko-build/setup-ko@ace48d793556083a76f1e3e6068850c1f4a369aa #v0.6

    - name: Build and push with ko
      id: build
      run: |
        # Build & push the image. Save the image name & digest
        image_and_digest=$(ko build --tags="${tag}" --bare --sbom=none .)

        # Output the image name and digest so we can generate provenance.
        digest=$(echo "${image_and_digest}" | cut -d'@' -f2)

    # digest/hash and image to outputs
        echo "digest=$digest" >> "$GITHUB_OUTPUT"
        echo "image=$KO_DOCKER_REPO" >> "$GITHUB_OUTPUT"
```

Pipeline - Sign Image

```
38    sign-image:
39      runs-on: ubuntu-latest
40      needs: [build]
41      permissions:
42        packages: write
43        id-token: write
44      env:
45        image: ${{ needs.build.outputs.image }}
46        digest: ${{ needs.build.outputs.digest }}
47      steps:
48        - name: Install cosign
49        - name: Login to ghcr.io
50
51        - name: Sign image
52          run: |
53            cosign sign "${image}@${digest}" --yes
```

Pipeline - Attach SBOM

```yaml
  sbom:
    runs-on: ubuntu-latest
    needs: [build]
    permissions:
      packages: write
      id-token: write
    env:
      image: ${{ needs.build.outputs.image }}
      digest: ${{ needs.build.outputs.digest }}
    steps:
      - name: checkout repo
      - name: Install cosign
      - name: Install Syft
      - name: Login to ghcr.io

      - name: Attach SBOM to image
        run: |
          # syft pulls the image and analyses the contents to generate an SBOM
          syft "${image}@${digest}" --output spdx-json --file sbom.spdx.json
          cosign attest --predicate sbom-final.spdx.json --type spdxjson "${image}@${digest}" --yes
```

Pipeline - Generate & Attach SLSA Provenance

```yaml
  # slsa-github-generator creates and pushes the provenance attestation
  provenance:
    needs: [build]
    permissions:
      actions: read
      id-token: write
      # contents: read
      packages: write
    if: startsWith(github.ref, 'refs/tags/') # just to be safe, don't push if it's not a tag
    uses: slsa-framework/slsa-github-generator/.github/workflows/generator_container_slsa3.yml@v1.9.0
    with:
      image: ${{ needs.build.outputs.image }}
      digest: ${{ needs.build.outputs.digest }}
      registry-username: ${{ github.actor }}
      compile-generator: true
    secrets:
      registry-password: ${{ secrets.GITHUB_TOKEN }}
```

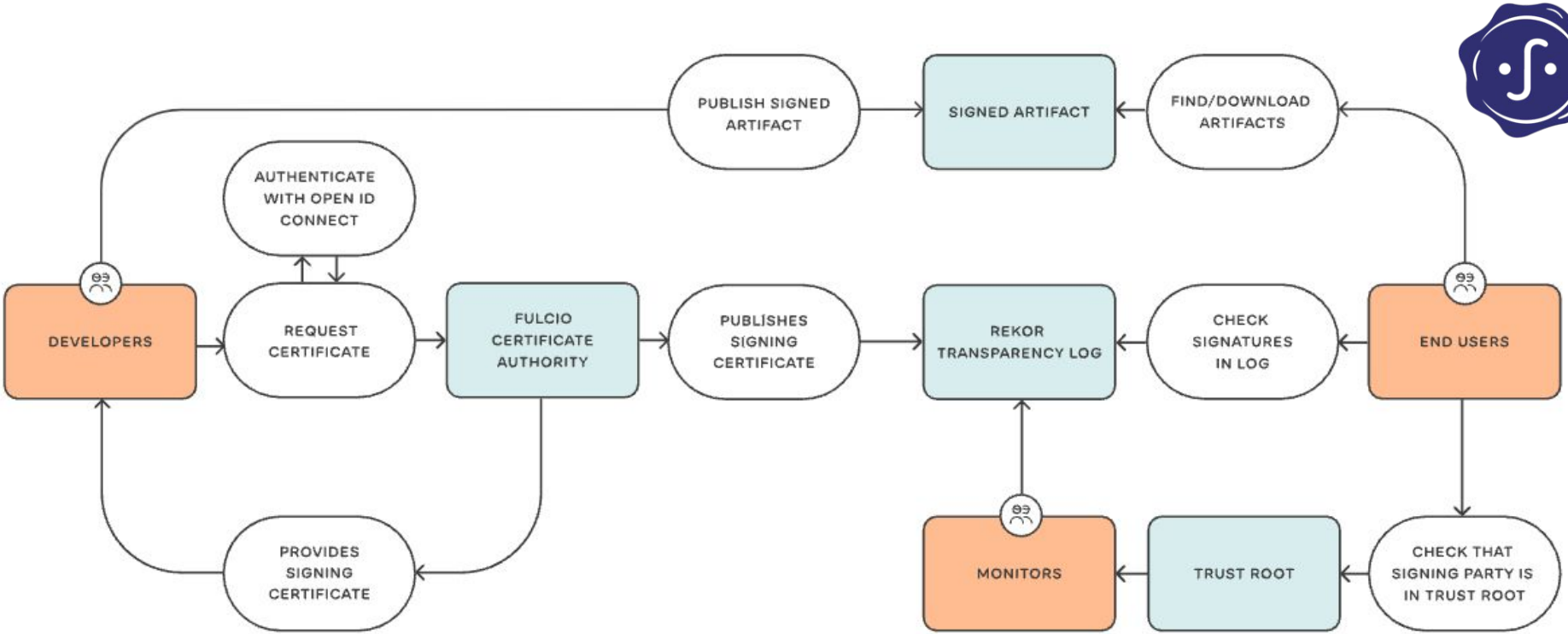
Open-source vs Private

Achieving SLSA (even Level 3) is fairly easy today for open-source projects, if you use the public sigstore instance and GitHub.

For private/proprietary projects, it's hard as you have to setup a code signing infrastructure. (also, how will consumers access it for verification?)

Signing

Sigstore



Source: https://www.sigstore.dev/how-it-works

Verifications

- Image Signature
- SBOM
- SLSA Provenance

Verifying

Verifying the Image with Nerdctl / Finch

```
finch run -it \
      --verify=cosign \
      --cosign-certificate-identity=https://github.com/chainguard-images/images/.github/workflows/release.yaml@refs/heads/main \
      --cosign-certificate-oidc-issuer=https://token.actions.githubusercontent.com \
      cgr.dev/chainguard/busybox /bin/sh
```

Verifying and downloading the SBOM (+grype)

```
cosign verify-attestation ghcr.io/verifa/verinotes:v0.2.0 \
    --certificate-identity-regexp '^https://github.com/verifa/verinotes/.github/workflows/.*.yaml@refs/tags/v0.*.*' \
    --certificate-oidc-issuer https://token.actions.githubusercontent.com \
    --type spdxjson | jq '.payload | @base64d' -r | jq '.predicate' | grype -v
```

```
    --type spdxjson | jq '.payload | @base64d' -r | jq '.predicate' | grype -v
[0000]  INFO grype version: 0.64.1
[0000]  INFO new version of grype is available: 0.73.3 (currently running: 0.64.1)
[0000]  INFO downloading new vulnerability DB

Verification for ghcr.io/verifa/verinotes:v0.2.0 --
The following checks were performed on each of these signatures:
  - The cosign claims were validated
  - Existence of the claims in the transparency log was verified offline
  - The code-signing certificate was verified using trusted certificate authority certificates
Certificate subject: https://github.com/verifa/verinotes/.github/workflows/release.yaml@refs/tags/v0.2.0
Certificate issuer URL: https://token.actions.githubusercontent.com
GitHub Workflow Trigger: push
GitHub Workflow SHA: d0a495e16d32c2bb125bf1481f747e711714199b
GitHub Workflow Name: Release workflow
GitHub Workflow Repository: verifa/verinotes
GitHub Workflow Ref: refs/tags/v0.2.0
[0001]  WARN some package(s) are missing CPEs. This may result in missing vulnerabilities. You may autogenerate these using: --add-cpes-if-none
[0010]  INFO updated vulnerability DB from version=5 built="2023-08-16 01:26:37 +0000 UTC" to version=5 built="2023-11-21 01:29:05 +0000 UTC"
[0010]  INFO found 9 vulnerabilities for 300 packages
NAME           INSTALLED  FIXED-IN  TYPE  VULNERABILITY        SEVERITY
@sveltejs/kit  1.6.0      1.15.1    npm   GHSA-5p75-vc5g-8rv2  High
@sveltejs/kit  1.6.0      1.15.2    npm   GHSA-gv7g-x59x-wf8f  High
postcss        8.4.21     8.4.31    npm   GHSA-7fh5-64p2-3v2j  Medium
semver         7.3.8      7.5.2     npm   GHSA-c2qf-rxjj-qqgw  Medium
undici         5.18.0     5.19.1    npm   GHSA-5r9g-qh6m-jxff  Medium
undici         5.18.0     5.19.1    npm   GHSA-r6ch-mqf9-qc9w  High
undici         5.18.0     5.26.2    npm   GHSA-wqq4-5wpv-mx2g  Low
vite           4.1.1      4.1.5     npm   GHSA-353f-5xf4-qw67  High
word-wrap      1.2.3      1.2.4     npm   GHSA-j8xg-fqg3-53r7  Medium
```

Verifying SLSA provenance

```
cosign verify-attestation \
  --type slsaprovenance \
  --certificate-oidc-issuer https://token.actions.githubusercontent.com \
  --certificate-identity-regexp '^https://github.com/slsa-framework/slsa-github-generator/.github/workflows/generator_container_slsa3.yml
  --policy policy.cue \
  ghcr.io/verifa/verinotes:v0.2.0 | jq '.payload | @base64d' -r | jq '.predicate'
```

File: **policy.cue**

```
// The predicateType field must match this string
predicateType: "https://slsa.dev/provenance/v0.2"

predicate: {
  // This condition verifies that the builder is the builder we
  // expect and trust. The following condition can be used
  // unmodified. It verifies that the builder is the container
  // workflow.
  builder: {
    id: =~"^https://github.com/slsa-framework/slsa-github-generator/.github/workflows/gener
  }
  invocation: {
    configSource: {
      // This condition verifies the entrypoint of the workflow.
      // Replace with the relative path to your workflow in your
      // repository.
      entryPoint: ".github/workflows/release.yaml"

      // This condition verifies that the image was generated from
      // the source repository we expect. Replace this with your
      // repository.
      uri: =~"^git\\+https://github.com/verifa/verinotes@refs/tags/v[0-9]+.[0-9]+.[0-9]+$"
    }
  }
}
```

```
cosign verify-attestation \
  --type slsaprovenance \
  --certificate-oidc-issuer https://token.actions.githubusercontent.com \
  --certificate-identity-regexp '^https://github.com/slsa-framework/slsa-github-generator/.github/workflows/generator_container_slsa3.yml@refs
  --policy policy.cue \
  ghcr.io/verifa/verinotes:v0.2.0 | jq '.payload | @base64d' -r | jq '.predicate'
will be validating against CUE policies: [policy.cue]

Verification for ghcr.io/verifa/verinotes:v0.2.0 --
The following checks were performed on each of these signatures:
  - The cosign claims were validated
  - Existence of the claims in the transparency log was verified offline
  - The code-signing certificate was verified using trusted certificate authority certificates
Certificate subject: https://github.com/slsa-framework/slsa-github-generator/.github/workflows/generator_container_slsa3.yml@refs/tags/v1.9.0
Certificate issuer URL: https://token.actions.githubusercontent.com
GitHub Workflow Trigger: push
GitHub Workflow SHA: d0a495e16d32c2bb125bf1481f747e711714199b
GitHub Workflow Name: Release workflow
GitHub Workflow Repository: verifa/verinotes
GitHub Workflow Ref: refs/tags/v0.2.0
{
  "builder": {
    "id": "https://github.com/slsa-framework/slsa-github-generator/.github/workflows/generator_container_slsa3.yml@refs/tags/v1.9.0"
  },
  "buildType": "https://github.com/slsa-framework/slsa-github-generator/container@v1",
  "invocation": {
    "configSource": {
      "uri": "git+https://github.com/verifa/verinotes@refs/tags/v0.2.0",
      "digest": {
        "sha1": "d0a495e16d32c2bb125bf1481f747e711714199b"
      },
      "entryPoint": ".github/workflows/release.yaml"
    },
    "parameters": {},
    "environment": {
      "github_actor": "mvainio-verifa",
```

There is also an official slsa-verifier

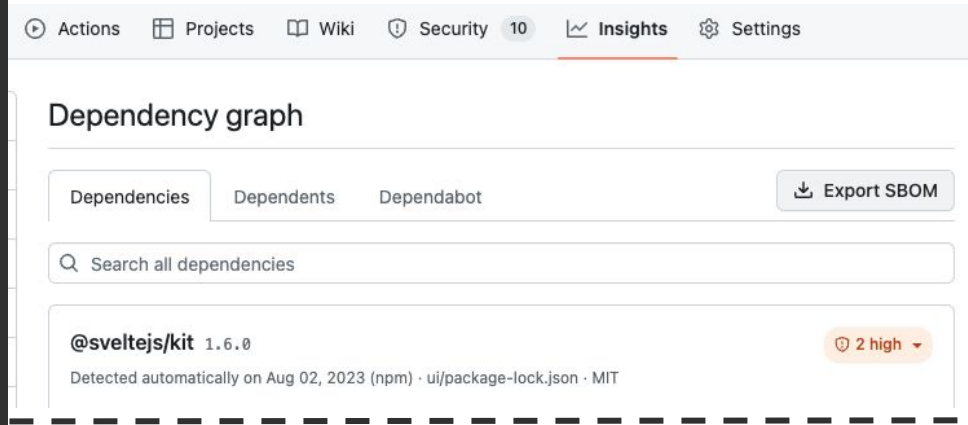
Kyverno / Policy Controller

Isn’t there an easier way?

GOOD NEWS!

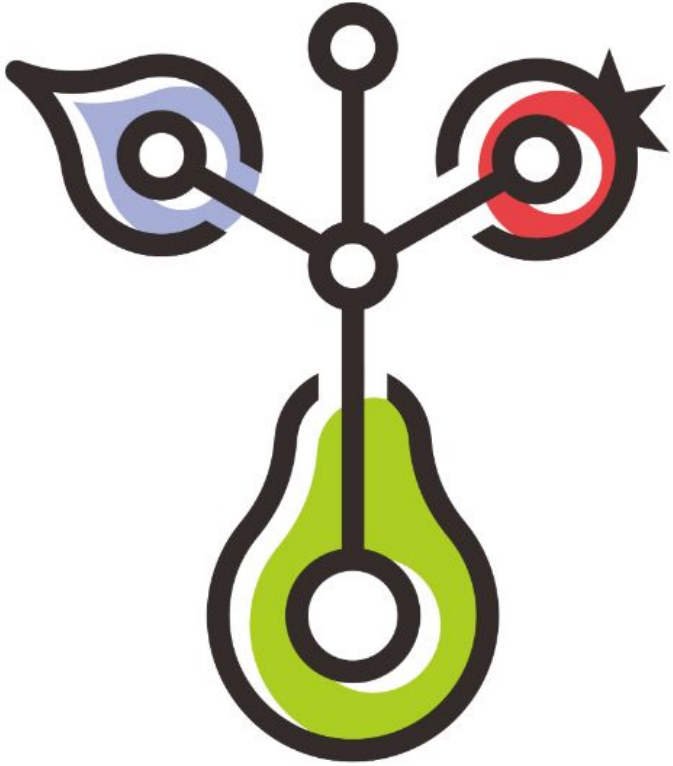
Platforms and open-source ecosystems are coming up with native integrations

GitHub



npm

I created this metadata, now what?



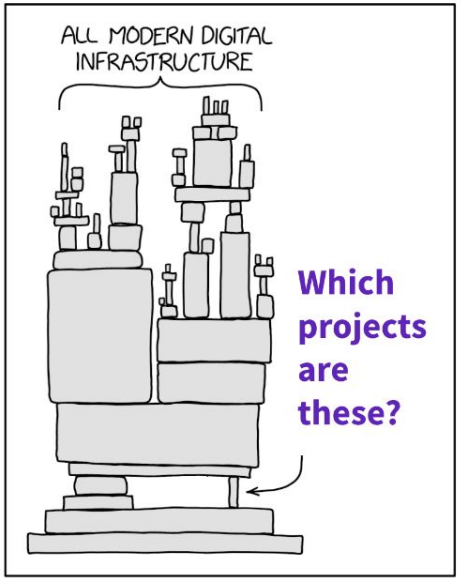
GUAC

Know your software supply chain

GUAC gives you directed, actionable insights into the security of your software supply chain.

Proactive

How do I prevent large scale supply chain compromises?



https://xkcd.com/2347/

Preventive

Have I taken the right safeguards?

When deciding to use and deploy software, are there sufficient security checks and approvals?

SLSA

trivy

grype

Reactive

HOW AM I AFFECTED???

A vulnerability or supply chain compromise is discovered!

Log4Shell™

Codecov

Check it out: https://github.com/guacsec/guac

Public Record

GitHub

OSV | Open Source Vulnerabilities

SLSA, SBOM, OSV, VEX, etc.

Org. (Private)

Azure Blob Storage

Google Cloud Storage

Amazon S3

SLSA, SBOM, Internal certification, etc.

Vendors

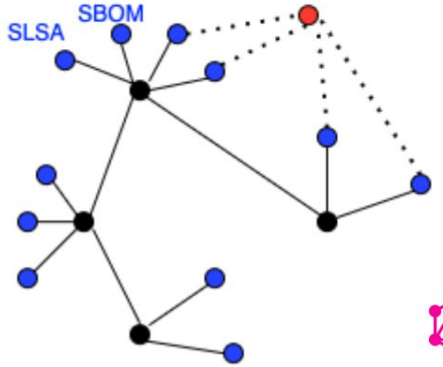
trivy

grype

Vuln certifications, OSV, VEX, etc.

Extract information and relationships

Ingest/ Process



Query

GraphQL

CISO
Inventory
Policy
CMDB

Graph contains:

- Artifacts/identifiers
- Attestations
- Identities
- Relationships (edges)

Further reading

What is in-toto and how it relates to SLSA?
https://slsa.dev/blog/2023/05/in-toto-and-slsa

OpenPubKey vs sigstore (note: by sigstore maintainer):
https://blog.sigstore.dev/openpubkey-and-sigstore/

OpenVEX (& VEX in general):
https://github.com/openvex

Homebrew core going for SLSA Build level 2:
https://blog.trailofbits.com/2023/11/06/adding-build-provenance-to-homebrew/

CRI-O support for verifying image signatures:
https://kubernetes.io/blog/2023/06/29/container-image-signature-verification/

OpenSSF blog/github etc.
https://openssf.org/blog/