SLSA — язык доверия

от CI до Runtime

Рыбалка Дмитрий
Principal SRE · Mindbox

ОПЫТ
SberHealth · SberMarket · Lamoda

СЕЙЧАС
Платформа и SRE

Говорю о том, что сам строю, внедряю и поддерживаю.

Мы уже многое делаем. На что именно это отвечает?

mindbox

ЧТО У НАС УЖЕ ЕСТЬ

- **Сборка в CI/CD**
- **Сканирование на уязвимости**
- **Подпись образов**
- **Policy checks**

НА ЧТО НЕ ОТВЕЧАЕТ

? **Кто собрал артефакт**

? **Из чего он собран**

? **В какой среде**

? **Где это проверяется перед запуском**

Trivy: что сломалось

mindbox

01 Residual access после неполной ротации credentials

02 Подмена тегов — force-push **76** из **77** тегов в trivy-action

03 Вредоносный релиз через доверенный pipeline

04 Потребители ниже по цепочке исполнили код атакующего в своих CI

Доверенный release path стал каналом доставки

Источник: aquasec.com/blog/trivy-supply-chain-attack-what-you-need-to-know

Где в Trivy ломается доверие

mindbox

Что случилось	Разрыв доверия	Защитная мера/Что помогает
Поверили тегу как якорю	Тег это указатель, его можно переставить	Привязка к полному хешу + проверка у потребителя
Доверились релизному пути без проверки происхождения	Подпись была, но без контекста: кто, откуда, через какой pipeline	Подписанное происхождение + сверка с ожидаемым pipeline
После первой компрометации сменили не все доступы	Долгоживущие учётки + неполная ротация	Короткоживущие доступы + изоляция сборки (L3)
Пайплайны потребителей запустили чужой код	Никто не проверял, что запускают	Проверка в двух точках — в CI и при запуске

Где расходится язык **платформы и ИБ**

mindbox

ПЛАТФОРМА

доказывает, что процесс прошёл

- **Pipeline зелёный**
- **Образ подписан**
- **Проверили в CI**

ИБ

спрашивает, можно ли принять policy-решение

? **Происхождение доказуемо?**

? **Кем, где, при каких условиях?**

? **Где enforcement(проверяется)?**

Это не спор о правоте. Это разговор на разных языках.
SLSA — попытка дать общий язык

Три слова, на которых держится модель SLSA

mindbox

01

Claims

что мы утверждаем

образ собран из commit abc123 через pipeline X

02

Evidence

чем подтверждаем

подписанный provenance с identity процесса сборки

03

Enforcement

система решает о запуске

admission controller проверяет signature перед запуском

L1: provenance — уровень видимости

SLSA spec version 1.2

Машиночитаемое утверждение о том, как был собран артефакт.

runDetails.builder.id	доверенная build platform/кто собрал
buildDefinition.buildType	какой build-процесс выполнялся
`buildDefinition.externalParameters`	какой pipeline, с какими параметрами
`buildDefinition.resolvedDependencies[]`	source commit и материалы сборки
`runDetails.metadata.startedOn/finishedOn`	время сборки

L1 может быть incomplete и unsigned — это видимость, не защита.
Provenance ≠ SBOM (опись содержимого) · ≠ build log (запись действий)

mindbox

GitLab CI: L1 — одна переменная

mindbox

Runner начинает генерировать SLSA L1 provenance для artifacts сборки.

```
# .gitlab-ci.yml
build-image:
  variables:
    RUNNER_GENERATE_ARTIFACTS_METADATA: "true"
```

На уровне Project / Group / Server.

Источник: https://docs.gitlab.com/ci/pipeline_security/slsa/

GitHub Actions

mindbox

Один step после сборки. Подпись через Sigstore с short-lived cert.
Это producer-side половина L2. Защита появляется только когда consumer реально проверяет attestation.

```yaml
# .github/workflows/release.yml
permissions:
  id-token: write  # OIDC для Sigstore
  attestations: write  # Хранить attestation
  contents: read
steps:
- name: Build
   …
- name: Generate build provenance attestation
  uses: actions/attest@59d89421af93a897026c735860bf21b6eb4f7b26 # v4.1.0
  with:
    subject-name: ${{ env.IMAGE }}
    subject-digest: ${{ steps.push.outputs.digest }}
    push-to-registry: true
    create-storage-record: false
```

Источник: https://docs.github.com/actions/security-for-github-actions/using-artifact-attestations/using-artifact-attestations-to-establish-provenance-for-builds

BuildKit: L1 — флаг --provenance

mindbox

Универсальный вариант — работает в любой CI. Один флаг.

```
# Минимум — провенанс по умолчанию (mode=min)
docker buildx build --provenance=mode=min -t myimage:tag --push .

# Полная картина — Dockerfile, layers, sourcemaps
docker buildx build --provenance=mode=max \
  -t myimage:tag --push .

# Привязка к build platform identity
--attest type=provenance,mode=max,builder-id=...
```

Внимание для mode=max: в provenance попадают значения build-args. Секреты передавай через `--secret`, не через build-args.

Источник: docs.docker.com/build/metadata/attestations/slsa-provenance

Включил флаг — это ещё L1

mindbox

Три способа ранее дают provenance(доказательство). Этого достаточно для L1.
Для L2 нужны три условия одновременно:

УСЛОВИЕ 1

Hosted build platform

Сборка на выделенной инфраструктуре, не на dev-машине разработчика/аналитика

УСЛОВИЕ 2

Signed provenance

Подпись привязана к этой платформе через её цифровую подпись

УСЛОВИЕ 3

Consumer verify

Клиент **обязательно** проверяет подпись.
Без этого цикла - защиты нет

GitHub Actions с attest-build-provenance закрывает условия 1 и 2 из коробки.
GitLab и BuildKit — только 1 по умолчанию, для 2 нужна дополнительная подпись.
Условие 3 — на стороне потребителя, а не платформы сборки.

Источник: https://slsa.dev/spec/v1.0/levels

L2: signed provenance — verifiable evidence

mindbox

Платформа сборки сама генерирует и подписывает provenance.

01 · IMAGE SIGNATURE

Байты артефакта не изменены

Подтверждает целостность образа. Подписывается ключом, привязанным к pipeline

02 · PROVENANCE ATTESTATION

Кем и как собран — с identity билдера

Подтверждает происхождение. Может иметь отдельный ключ и жизненный цикл.

Частая ошибка: сторона выпуска настроена, подпись создается, attestation подписана, хранится в registry — но никто её не проверяет.
Это не L2. Verify со стороны consumer — обязательная часть L2 по spec.

Источник: aquasec.com/blog/trivy-supply-chain-attack-what-you-need-to-know

В закрытой инфре: cosign + Vault

mindbox

Sigstore с публичным Fulcio часто не подходит — air-gapped(изолированная среда), регуляторика, своя политика.
Cosign работает с любым KMS, включая Vault.

```
# Предусловие — на стороне Vault
vault secrets enable transit
cosign generate-key-pair --kms hashivault://release-key

# Подпись artifact и provenance attestation
cosign sign --key hashivault://release-key $IMAGE
cosign attest --predicate provenance.json --type slsaprovenance1 \
    --key hashivault://release-key \
    --tlog-upload=false $IMAGE
# На стороне consumer
cosign verify-attestation --insecure-ignore-tlog=true \
 --insecure-ignore-sct=true \
 --key hashivault://release-key --type slsaprovenance1 $IMAGE
```

Приватный ключ не покидает Vault (Transit engine). `--tlog-upload=false` — без публичного Rekor. Verify работает без выхода в интернет, нужен только public key.

Сторона выпуска: утверждения и доказательства

mindbox

Артефакт выпуска

01 Release identity привязана к CI pipeline, не к человеку

02 Provenance для каждого release build

03 Подпись артефакта, и provenance attestation

> Для зала это база, но запись смотрят разные команды. Аналитики и ML-инженеры часто собирают локально. Глубже — это вопрос не "откуда собирают", а кто имеет право подписать release, локальная сборка не сможет выпустить артефакт как release, если право подписи привязано к pipeline.

Producer даёт claims. Без consumer-side это ещё не trust chain.

Сторона потребления: доказательство становится решением

mindbox

Что делает команда, использующая артефакт

01 Digest-first вместо тегов

02 Отдельный verify step в pipeline

03 Audit before enforce — сначала логируем, потом блокируем

```
# UNSAFE — mutable tag
uses: aquasecurity/trivy-action@0.30.0

# SAFE — pinned to immutable commit SHA
uses: aquasecurity/trivy-action@57a97c7e7821a5776cebc9bb87c984fa69cba8f1
```

Граница доверия — свойство среды

mindbox

Что ослабляет среду:

- Общие ранеры с общим хранилищем
- RW кеш, общий между jobs
- Долго живущие секреты в переменных сборки
- Лишние загрузки из вне
- Один builder.id на множество пайплайнов

AQUA О СВОЁМ COMMERCIAL ENVIRONMENT

Архитектурно изолированный pipeline, Отдельные репозитории, отдельные secrets, отдельная инфраструктура сборки, gated security review перед интеграцией изменений из upstream.

Принудительная проверка: — от CI до Runtime

mindbox

CI consumption

ЧТО ЗАПУСКАЕТСЯ ВО ВРЕМЯ СБОРКИ

- Сторонние actions закреплены SHA
- Разрешены только допустимые actions
- Обязательная проверка для build tools
- Все то что не проверено, запрещено

Runtime consumption

ЧТО ЗАПУСКАЕТСЯ В КЛАСТЕРЕ

- Запуск с привязкой к SHA
- Admission policy проверяет подпись
- Проверяется builder id и source commit, Subject привязан к ожидаемому release
- Все что не проверено - не запускается

Один принцип. Две точки применения.

ENFORCEMENT

Kyverno: проверка provenance в кластере

mindbox

Нативная поддержка cosign. Подпись + поля provenance — в одной policy.

```
# Проверяем подпись и поля provenance attestation
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: verify-provenance-from-trusted-builder
spec:
  validationFailureAction: Enforce
  rules:
  - name: check-builder-and-source
    match: { any: [{ resources: { kinds: [Pod] }}]}
    verifyImages:
    - imageReferences: ["registry.corp.local/*"]
      attestors:
      - entries:
        - keys: { publicKeys: "|- -----BEGIN PUBLIC KEY----- MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE.." }
      attestations:
      - type: https://slsa.dev/provenance/v1
        conditions:
        - all:
          - key: "{{ predicate.buildDefinition.externalParameters.source }}"
            operator: Equals
            value: “git+https://git.corp.local/platform/myapp"
          - key: "{{ predicate.runDetails.builder.id }}"
            operator: Equals
            value: “https://gitlab.corp.local/runner@release"
```

ГРАНИЦЫ SLSA

Что SLSA делает и не делает

mindbox

SLSA не убирает supply-chain как класс атак.

SLSA делает происхождение видимым, подписанным и пригодным для policy.

ОСТАЁТСЯ ЗА ПРЕДЕЛАМИ

Malicious maintainer

человек с легитимным правом подписи может злоупотребить

Code bugs

уязвимости в самом коде зависимостей

Compromised identity

если credentials украдены — подпись валидна. Но L3 снижает blast radius через разделение секретов для процесса сборки и подписи

Что Aqua сделали после инцидента

mindbox

Список мер из их post-mortem от апреля 2026:

- Pin GitHub Actions к специфичным commit SHAs
- Immutable releases в Docker Hub и GitHub Releases
- Git tag protection для artifact repositories
- **Added SLSA provenance attestations to Trivy releases**
- SSO и IP allow-listing across the organization
- Сокращение long-lived credentials, переход к scoped tokens

Это то, что мы только что разобрали.
Список который стал политикой после инцидента**.**

Источник: https://github.com/aquasecurity/trivy/discussions/10462

mindbox

Build succeeded — лог события.

Signed — проверяемый claim.

Доверие — там, где claim влияет на решение.

Неподтверждённый артефакт не должен попадать в execution path

Обсудим

1. SLSA L1/L2/L3
2. SLSA + Enforcement
3. SLSA + air-gapped
4. SLSA + в Opensource

Фидбек

1 вопрос

mindbox БЕКОН26

Backup · SLSA .уровни кратко

mindbox

L1

Provenance существует и публикуется

Закрывает: базовая отслеживаемость, расследование

L2

Provenance подписан, hosted build service

Закрывает: tampering с provenance, подделка от имени builder

L3

Build platform изолирована, identity нельзя подделать

Закрывает: компрометация build environment, side-channel

L4 в текущей версии(v1.0) не выделяется как отдельный уровень

Backup · Четыре разных объекта

SBOM

Опись содержимого: какие пакеты, версии, лицензии внутри артефакта

Отвечает на: что внутри

Provenance

Машиночитаемое утверждение о том, как был собран артефакт

Отвечает на: как появился

Signature

Криптографическое подтверждение целостности артефакта или утверждения

Отвечает на: не изменён ли

Attestation

Подписанное утверждение о свойстве артефакта (provenance это вид attestation)

Отвечает на: что и кто заявляет

Эти объекты дополняют друг друга, не заменяют

Backup · Что и как закрывает SLSA

mindbox

ЗАКРЫВАЕТ

— Подмена артефакта между сборкой и потреблением
— Tampering с tag → редирект на malicious commit
— Подделка provenance от имени builder
— Использование артефакта из недоверенного pipeline

НЕ ЗАКРЫВАЕТ

— Malicious maintainer с легитимным правом подписи
— Уязвимости в коде самих зависимостей
— Кража credentials у builder identity
— Социальная инженерия на этапе review

Backup - **Advanced: compact predicate для admission**

“Это не замена SLSA provenance. Это компактный policy-signal для runtime.

```
# 1. Создаем минимальный predicate — 7 полей, ~500 байт
printf '%s\n' \
  '{' \
          '    "pipeline_url": "'"${CI_PIPELINE_URL}"'",' \
          '    "pipeline_id": "'"${CI_PIPELINE_ID}"'",' \
          '    "job_url": "'"${CI_JOB_URL}"'",' \
          '    "job_id": "'"${CI_JOB_ID}"'",' \
          '    "project_url": "'"${CI_PROJECT_URL}"'",' \
          '    "ref": "'"${CI_COMMIT_REF_NAME}"'",' \
          '    "sha": "'"${CI_COMMIT_SHA}"'"' \
          '  }' \

  '}' > ci-invocation.json

# 2. Подписываем через Vault, без публичного Rekor
```

Разделение concerns: полный SLSA provenance (с buildConfig, layers, sourcemaps) остаётся в registry — для аудита, compliance, расследования инцидентов. Custom ci-invocation — только для admission в проде. Kyverno проверяет именно его.

Требования SLSA v1.0: L0 → L1 → L2 → L3

Backup-материал к докладу «SLSA — язык доверия: от CI до Runtime».

Используется для ответа на популярный gotcha-вопрос: «включил флаг — получил уровень?»

Источник: slsa.dev/spec/v1.0/levels

mindbox

Включил флаг — получил уровень?

ТИПИЧНЫЙ ВОПРОС ИЗ ЗАЛА

«Я добавил `actions/attest-build-provenance@v1` в workflow — теперь у меня L2?»

КОРОТКИЙ ОТВЕТ

Нет. Это часть L1.

Включение генерации provenance — необходимое, но недостаточное условие.
SLSA-уровни описывают **сочетание свойств** producer'a, build platform и consumer'a.

mindbox

L0 → L1 → L2 → L3: от чего защищает каждый

Уровни — не «больше галочек», а **разные классы защиты**. Каждый закрывает свой тип атаки.

УРОВЕНЬ	ОТ ЧЕГО ЗАЩИЩАЕТ	ГЛАВНОЕ ТРЕБОВАНИЕ
L0	Ничего	Нет requirements — отсутствие SLSA
L1	Mistakes, документация	Provenance существует и публикуется
L2	Tampering **после** сборки	Signed provenance + hosted build platform
L3	Tampering **во время** сборки	Hardened build platform

Главное: tampering после сборки (L2) и tampering во время сборки (L3) — это разные классы атак. L3 — не «лучше чем L2», это **дополнительная защита от другого вектора**.

mindbox

L1: три требования

Все три должны выполняться. На L1 provenance может быть **incomplete и/или unsigned** — спека это явно разрешает.

ТРЕБОВАНИЕ 1

Consistent build process

Producer следует повторяемому процессу, чтобы сторонний верификатор мог сформировать ожидания о том, как выглядит «правильная» сборка.

ТРЕБОВАНИЕ 2

Provenance существует

Содержит build platform, build process и top-level inputs. Это закрывается включением флагов в GitLab/GitHub/BuildKit.

ТРЕБОВАНИЕ 3

Distribution потребителям

Producer публикует provenance через convention экосистемы (npm, OCI registry, etc). Без распространения — нет L1.

Частое заблуждение: «включил `RUNNER_GENERATE_ARTIFACTS_METADATA` → готово». Это закрывает только требование 2. Требования 1 и 3 — отдельная работа.

mindbox

L2: всё из L1 + три условия

Producer и consumer вместе — без обеих сторон L2 не работает.

К BUILD PLATFORM

Hosted infrastructure

Build platform запускается на dedicated infrastructure. **Self-hosted runner на VM разработчика — НЕ L2.**

К BUILD PLATFORM

Digital signature

Provenance подписана digital signature, привязанной к этой инфраструктуре. Не разработчиком, не CI-токеном — платформой.

К CONSUMER

Verify authenticity

Downstream verification **обязательно** включает validating authenticity подписи. **Без этого L2 не существует.**

Самая частая ошибка: producer-side настроен идеально — но никто не делает `gh attestation verify`. **Это не L2.** Подпись существует, но цикл защиты разомкнут.

mindbox

L3: всё из L2 + hardening build platform

Build platform реализует strong controls. Это **архитектурная работа**, не флажок в CI.

КОНТРОЛЬ 1

Изоляция между runs

Один run не может влиять на другой даже в рамках одного проекта. Никакого shared state, mutable cache, persistent FS между запусками.

КОНТРОЛЬ 2

Защита signing material

Secret material для подписи provenance недоступен из user-defined build steps. Ключ изолирован от кода, который собирается.

Что нарушает L3:

- **Composite action в GitHub** — выполняется в job вызывающего, не изолирован
- **Self-hosted runner как long-lived VM** с persistent state между запусками
- **Signing key в env variables CI** — доступен из user-defined steps
- Reusable workflows — наоборот, L3-совместимая граница (запускаются в отдельном job)

Три роли — три зоны ответственности

Producer **не может в одиночку** достичь L2 или L3. SLSA — это распределённая модель ответственности.

РОЛЬ	ЗА ЧТО ОТВЕЧАЕТ
Software Producer	Consistent build process, выбор build platform правильного уровня, distributing provenance потребителям.
Build Platform	Генерация provenance, подпись (L2+), изоляция runs (L3), защита signing material (L3). Это где живёт большая часть L3-требований.
Verifier / Consumer	Validation подписи (L2+), сравнение полей provenance с expected values. **Без consumer'a никакой реальный уровень не достигается.**

Что значит L2 — на конкретных платформах

GitHub Actions

`actions/attest-build-provenance@v1` + GitHub-hosted runner = **producer-side L2 ready**

L2 достигается **только если** кто-то реально вызывает `gh attestation verify` в pipeline или admission policy

L3 требует **reusable workflows**, не composite actions

GitLab CI

`RUNNER_GENERATE_ARTIFACTS_METADATA: "true"` даёт **только L1** (provenance без подписи)

L2 требует дополнительного CI/CD компонента с `cosign attest-blob` + Sigstore или внутренним KMS

L3 — gated runner pool, изоляция между jobs, отдельная signing identity

BuildKit

`docker buildx --provenance=mode=max` даёт **L1 по умолчанию** (без подписи)

L2 — добавь `cosign sign --key` поверх образа

L3 зависит от среды запуска самой buildx-сборки (где работает builder)