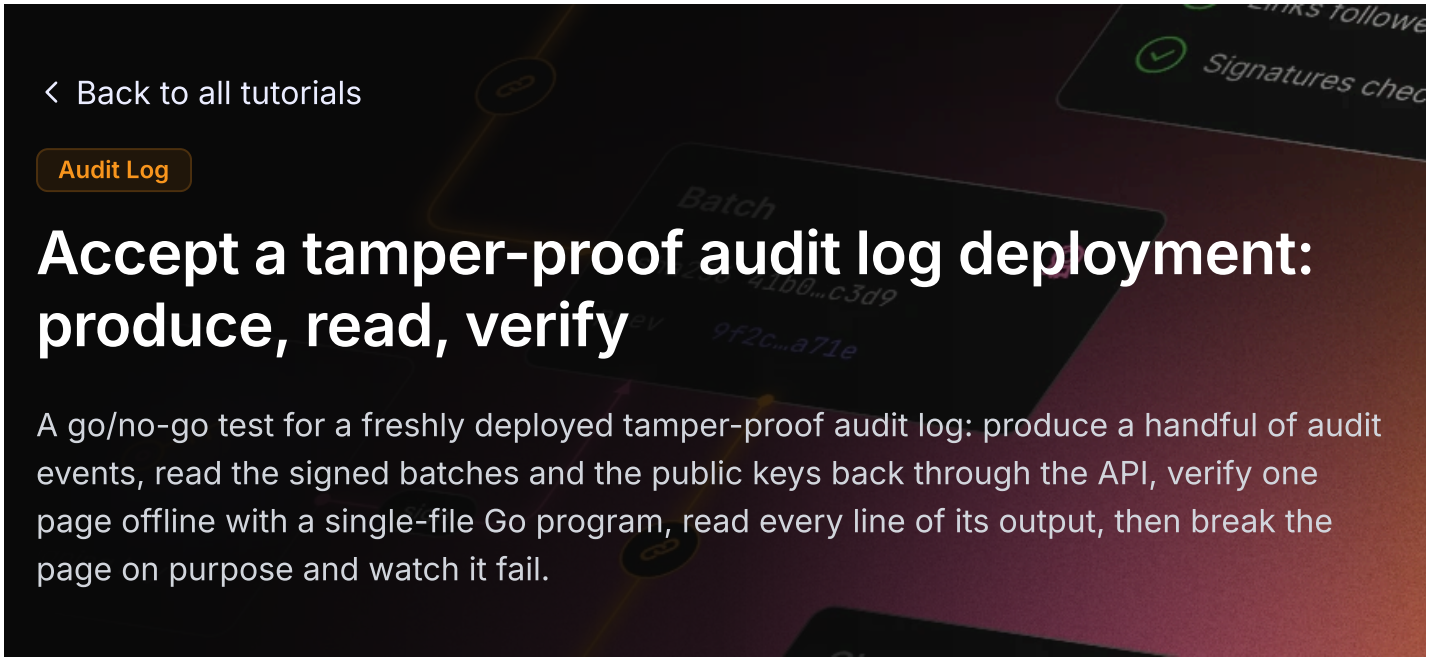
Source: https://developer.indykite.com/tutorials/tutorial-audit-acceptance · PDF generated 2026-11-25; the page is the current version.



A tamper-proof audit log is only worth its name once someone outside the platform has verified it. This tutorial is that first verification, written as an acceptance test: short enough to run in an afternoon, exact enough to be the gate before the one-way decisions of a dedicated deployment, such as locking the bucket's retention. It works just as well against the shared IndyKite platform, where the infrastructure is not yours but the proof still is.

It uses `curl`, `jq` and one Go file you download. Nothing here needs the Hub beyond the credentials you create there once.

What you will have at the end

1. Audit events of your own in the project's chain: Capture upserts and configuration reads.
2. Two saved API responses: a page of signed batches and the JWK Set that signed them.
3. A verifier you ran yourself, which recomputed every batch hash, checked every signature against the published keys and followed every chain link on the page.
4. The ability to read its output: what each failure line means, which ones point at the data and which at the keys.
5. Proof that the verifier bites: the same page altered by one character, and the lines that catch it.

Prerequisites

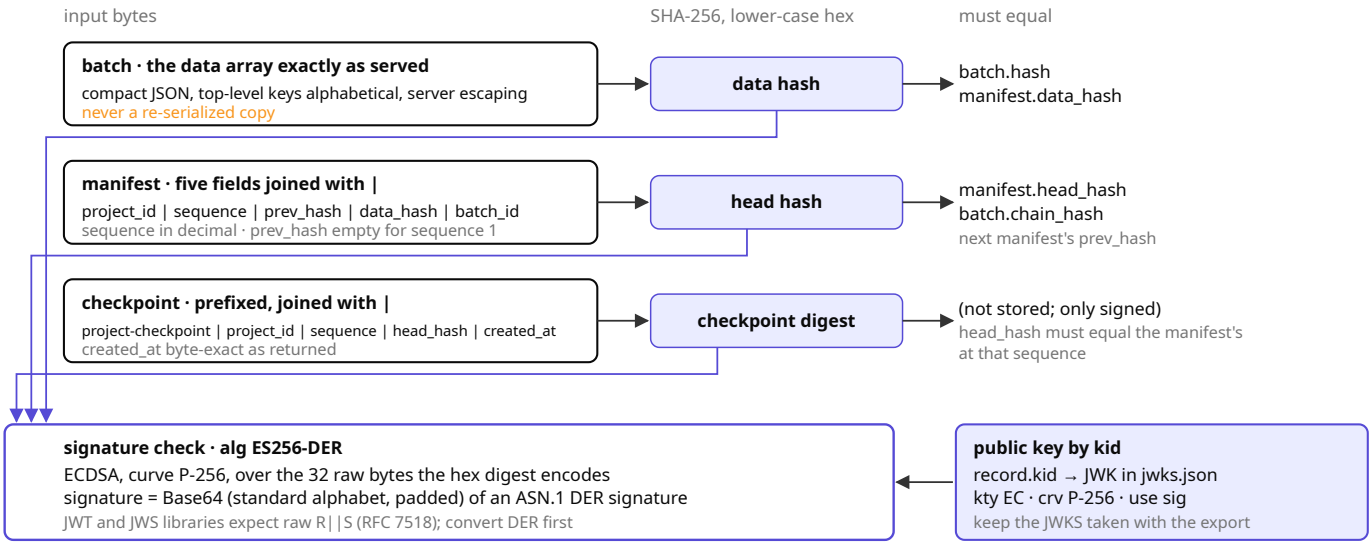
- An IndyKite **project** and its GID, a **Service Account** credential for the Config API, and an **Application Agent** credential holding the `Audit` and `Capture` API permissions. Chapter 1 lists where each comes from.
- A destination you own for an **Event Sink** (a Kafka topic, a Pub/Sub topic, an Event Grid topic or a Service Bus queue): the chain holds only the events a sink of the project delivered, so chapter 2 creates one, together with the Audit Signing configuration that switches the chain on.
- `curl`, `jq`, and **Go 1.25 or later** for chapter 4.

- For a dedicated deployment: the public API host of that deployment, and someone on the platform side who can tell you the integrity verifier's last exit code (chapter 5).

Who this tutorial is for

Platform and security engineers accepting a dedicated deployment of the audit log, as described in the **infrastructure guide**; compliance teams who want a repeatable spot check; and anyone who prefers a single Go file to a Python environment. For the full export, with manifests and checkpoints, and the operating habits around it, continue with the **audit trail tutorial**.

The flow at a glance



CHAPTERS

Chapter 1: What you need

Chapter 2: Produce events

Chapter 3: Read the trail and the keys

Chapter 4: Run the verifier

Chapter 5: Read the result, then break it

CHAPTER 1

1

What you need

The project, the two credentials and their permissions, the API host of a shared or dedicated deployment, and the shell variables the next chapters use.

Chapter 1: What you need

Four values and two tools. Everything else is created along the way.

Step 1: The project and its credentials

What	Used for	Where it comes from
Project GID	The `project_id` of every Audit Log API call.	The **environment setup example** creates one with the Config API; an existing project's GID is in its read response.
Service Account token	The Config API calls that produce configuration events (chapter 2).	The **Credentials guide**; a credential created with an `expire_time` carries a ready-to-use `token`.
Application Agent credential with `Audit` and `Capture`	`Capture` for the events of chapter 2, `Audit` for reading the trail in chapter 3. Add `Authorization` or `ContXIQ` too if you want decision or query events in the trail.	Set `api_permissions` when creating the agent, or add the values to an existing agent with `PUT /configs/v1/application-agents/{id}`; see the **Environment guide**.
API host	Every call.	Shared platform: `https://us.api.indykite.com` or `https://eu.api.indykite.com`, the region of the project. Dedicated deployment: the public host under your domain that the deployment exposes, for example `https://api.example.com`. The paths are identical.

An agent without `Audit` gets `401` and `insufficient API access level for appAgent` on every Audit Log endpoint except the JWK Set, which needs no permission.

Step 2: Export the variables

The next chapters use these names. Put them in a file you `source`, so the verifier's input files and the curl calls agree:

```
export API_URL=https://us.api.indykite.com   # or https://eu.api.indykite.com,
or https://api.<your-domain>
export PROJECT_ID="gid:YOUR_PROJECT_GID"
export SA_TOKEN="<token from the Service Account credentials file>"
export CLIENT_KEY="<token from the Application Agent credentials file>"
```

Check the agent before producing anything: a `200` here, even with an empty `items` array, means the host, the credential and the permission are right.

```
curl -sS -o /dev/null -w "%{http_code}\n" "$API_URL/audit/v1/logs?
project_id=$PROJECT_ID&pagesize=1" \
  -H "X-IK-ClientKey: $CLIENT_KEY" -H "Accept: application/json"
```

Step 3: The tools

- `curl` and `jq`, for the calls and for looking at what comes back.
- **Go 1.25 or later** (`go version`). The verifier is one file with no dependencies outside the standard library; it is run with `go run`, nothing is installed.

How this differs from the **audit trail tutorial**: that one exports a whole chain, manifests and checkpoints included, and verifies all of it with a Python script. This one verifies a single page of batches with a Go program and stops. It is the quick gate, not the archive.

CHAPTER 2

2

Produce events

Put a few audit events of each kind into the chain: Capture upserts with the sample dataset, and configuration reads and lists with the Config API.

Chapter 2: Produce events

A chain with nothing in it cannot be verified. First the two configurations the chain depends on, then two kinds of calls you would make anyway: a Capture ingest and a couple of Config API reads.

Step 1: Switch the chain on

An event is chained only when the project has an **Audit Signing** configuration and an **Event Sink** of the project delivered the event to at least one destination. Both are Config API objects; skip whichever the project already has.

```
curl -sS -X POST "$API_URL/configs/v1/audit-signings" \
  -H "Authorization: Bearer $SA_TOKEN" -H "Content-Type: application/json" \
  -d '{ "project_id": "'"$PROJECT_ID"'", "name": "audit-signing", "provider":
"PLATFORM_MANAGED" }'
curl -sS -X POST "$API_URL/configs/v1/event-sinks" \
  -H "Authorization: Bearer $SA_TOKEN" -H "Content-Type: application/json" \
  -d '{
    "project_id": "'"$PROJECT_ID"'",
    "name": "acceptance-sink",
```

```
    "providers": {
      "audit-kafka": {
        "kafka": {
          "brokers": ["broker.example.com:9094"],
          "topic": "indykite-audit",
          "username": "your-user",
          "password": "'"$KAFKA_PASSWORD"'"
        }
      }
    },
    "routes": [
      {
        "provider_id": "audit-kafka",
        "event_type_key_values_filter": { "event_type": "indykite.audit.*",
"context_key_value": [] }
      }
    ],
    "include_cdc_events": false
  }'
```

The sink's destination must accept deliveries: an event the destination rejects, or that no route matches, is not chained. The Outbound Events guide shows the Pub/Sub, Event Grid and Service Bus forms and how to read a provider's `last_error`. A project has one Event Sink; if one exists, add a route on `indykite.audit.*` to it instead of creating another.

Step 2: Capture the sample dataset

The two files are the ones the **Data Schema guide** uses: eight nodes and five relationships of a car-rental company.

- **simple-dataset-nodes.json**
- **simple-dataset-relationships.json**

```
curl -sS -X POST "$API_URL/capture/v1/nodes" \
  -H "X-IK-ClientKey: $CLIENT_KEY" -H "Content-Type: application/json" \
  --data @simple-dataset-nodes.json
curl -sS -X POST "$API_URL/capture/v1/relationships" \
  -H "X-IK-ClientKey: $CLIENT_KEY" -H "Content-Type: application/json" \
  --data @simple-dataset-relationships.json
```

Each node becomes one `indykite.audit.capture.upsert.node` event and each relationship one `indykite.audit.capture.upsert.relationship` event. Both calls are upserts, so running them again is harmless to the graph and adds another round of events, which is useful if you want more than one page later.

Step 3: Read and list configuration

Reads and lists are audited too, not only writes. Two Service Account calls that change nothing:

```
curl -sS "$API_URL/configs/v1/token-introspects?project_id=$PROJECT_ID" \
  -H "Authorization: Bearer $SA_TOKEN" -H "Accept: application/json" | jq '.data
| length'
curl -sS "$API_URL/configs/v1/projects/$PROJECT_ID" \
  -H "Authorization: Bearer $SA_TOKEN" -H "Accept: application/json" | jq
'{name, ikg_status}'
```

The list leaves an `indykite.audit.config.list` event carrying the IDs it returned, the read an `indykite.audit.config.read` event. If you also hold `Authorization` or `ContXIQ` on the agent, an AuthZEN evaluation or a Knowledge Query execution adds the `authorization.*` and `ciq.execute` families; the **Outbound Events guide** lists every event type.

Step 4: Wait for the batch

Events reach the chain in batches, cut by size or by time, so the first page may lag the calls by a short while. The next chapter's first call tells you when it is there: a non-empty `items` array whose last event is one you just produced.

CHAPTER 3

3

Read the trail and the keys

Save a page of signed batches and the JWK Set as two files, look at what a batch contains, and check that the key each batch names is one the platform publishes.

Chapter 3: Read the trail and the keys

Two calls, two files. Save the responses exactly as served, since the verifier recomputes hashes over the bytes of `data`; `curl -o` does that, a copy-paste through an editor may not.

Step 1: A page of batches

```
curl -sS "$API_URL/audit/v1/logs?project_id=$PROJECT_ID&pagesize=50" \
  -H "X-IK-ClientKey: $CLIENT_KEY" -H "Accept: application/json" \
  -o response-logs.json
jq '{batches: (.items | length), first: .items[0].sequence, last:
.items[-1].sequence, has_more, next_cursor}' response-logs.json
```

Batches come oldest first. A page is at most `pagesize` batches; `has_more` says whether there are more, and `next_cursor` is what you pass as `cursor=` to get them. For the acceptance test one page is enough; the **Audit Log guide** has the paging rules for a full export.

What one batch carries, and what the verifier will do with each field:

```
jq '.items[0] | {sequence, batch_id, kid, alg, hash, chain_hash, manifest_id,
events: (.data | length), first_event: .data[0].eventType}' response-logs.json
```

Field	Meaning	Checked how
`data`	The events of the batch, a JSON array, as the platform received them.	Its SHA-256 must equal `hash`.
`hash`, `signature`, `kid`, `alg`	The digest of `data`, its signature, the key that made it and the algorithm, always `ES256-DER`.	The signature must verify over `hash` under the JWK with that `kid`.
`sequence`, `chain_hash`, `batch_id`, `project_id`	The batch's position, the head of the chain after it, and the identifiers that go into the head.	`chain_hash` must equal the SHA-256 of `project_id|sequence|previous chain_hash|hash|batch_id`; sequences must be consecutive.
`manifest_id`	The manifest that records this batch in the chain.	Not on this page; the full export verifies manifests and checkpoints.

Step 2: The JWK Set

The public keys need no credential: anyone may verify.

```
curl -sS "$API_URL/audit/.well-known/jwks.json?project_id=$PROJECT_ID" -o
response-jwks.json
jq '.keys[] | {kid, kty, crv}' response-jwks.json
```

Each key is an EC P-256 public key with a `kid`. Before running anything, confirm the page and the keys talk about the same key:

```
jq -r '.items[].kid' response-logs.json | sort -u          # kids the page uses
jq -r '.keys[].kid' response-jwks.json | sort -u           # kids the platform
publishes
```

Every value of the first list must appear in the second. On a dedicated deployment the `kid` is the key identifier agreed at hand-over, the one the deployment was configured to stamp on every signature. If it is not that value, stop here; the deployment signs with a key, or a configuration, you did not intend.

Step 3: Keep the keys independently

A verification is only as trustworthy as its keys. Fetch the JWK Set yourself, from the API, and keep it with the export; never accept the keys from whoever hands you the export, since one party controlling both could have signed both. The `Cache-Control` header of the JWK Set response says how long the set may be reused.

CHAPTER 4

4

Run the verifier

Run the single-file Go verifier on the saved page and keys, understand the five checks it makes on every batch, and read its exit codes.

Chapter 4: Run the verifier

The verifier needs the two files from chapter 3 and nothing else: no credential, no network, no IndyKite. It is a single Go file using only the standard library.

Step 1: Run it

Download **audit-verify.go** next to the two files and run:

```
go run audit-verify.go -logs response-logs.json -jwks response-jwks.json
```

```
OK   seq=1 batch=bf76b15c-9fc6-49bb-adca-10622039f65c events=17
OK   seq=2 batch=311bbd81-6c0c-4ec1-9e8e-aac8d7a79033 events=2
OK   seq=3 batch=7d722889-e005-413c-af18-4ce0d6bfc03b events=2
…
OK   seq=29 batch=2106a710-eb39-4266-aca9-ea809ddf6465 events=1
29 batches, 0 failed
```

One line per batch, oldest first, with the number of events it holds, then a summary. The sequence numbers are yours; the first batch of a new project is `seq=1` and usually the largest, since the project's creation and setup left many events before the first batch was cut.

Step 2: What it checks, per batch

1. **The batch hash.** `data` is re-encoded as compact JSON, exactly the bytes the platform hashed, and its SHA-256 must equal `hash`. This is why the response must be saved as served: re-encoding restores the bytes even if the file was pretty-printed, but a changed character in any event changes the digest.
2. **The signature.** `alg` must be `ES256-DER`; the key named by `kid` must be in the JWK Set; and the DER-encoded ECDSA signature, Base64-decoded, must verify over the digest that `hash` stands for. A signature that verifies proves the platform's key produced it; the hash check then ties `data` to it.
3. **The chain link.** `chain_hash` must equal the SHA-256 of `project_id|sequence|previous chain_hash|hash|batch_id`, where the previous `chain_hash` is the one of the batch before. For the first batch on the page the previous value is empty when the page starts at sequence 1; otherwise pass it with `-prev`, or the first link is reported as not checked.

4. **Continuity.** Each sequence must be the previous one plus one: a missing or reordered batch is reported even when every batch verifies on its own.
5. **One project.** Every batch on the page must carry the same `project_id`.

The checks run on every batch and all problems are printed; one bad batch does not hide the state of the batches after it, because each link is checked against what the previous batch *records*, not against what was recomputed.

Step 3: Exit codes

Exit code	Meaning
`0`	Every batch on the page passed every check.
`1`	At least one batch failed a check; the lines above the summary say which and why.
`2`	The verifier could not run: a file is missing or not JSON, the JWK Set has no usable P-256 key, or the page has no items. Nothing was verified.

The exit code makes it a pipeline step: a nightly job that fetches one page and the keys and fails the build on anything but `0` is a reasonable first monitor, independent of the platform's own integrity verifier.

Step 4: A later page

To verify a page that does not start at sequence 1, take the `chain_hash` of the last batch of the page before it:

```
PREV=$(jq -r '.items[-1].chain_hash' response-logs.json)
CURSOR=$(jq -r '.next_cursor' response-logs.json)
curl -sS "$API_URL/audit/v1/logs?
project_id=$PROJECT_ID&pagesize=50&cursor=$CURSOR" \
  -H "X-IK-ClientKey: $CLIENT_KEY" -H "Accept: application/json" -o response-
logs-2.json
go run audit-verify.go -logs response-logs-2.json -jwks response-jwks.json -prev
"$PREV"
```

The last line of the summary tells you when you are done: it repeats `has_more` and the cursor to fetch next while the chain continues.

CHAPTER 5

5

Read the result, then break it

What each failure line of the verifier means and where to look, two deliberate alterations that make it fail, and what to do after the test passes: the full export, the platform-side checks, and the retention lock.

Chapter 5: Read the result, then break it

Step 1: The failure lines

A failed batch prints `FAIL` instead of `OK` and one indented line per problem. What each one means:

Line	Meaning	Where to look
`data hashes to …, batch records …`	The events are not the bytes that were signed.	The file was altered or re-serialised after download, or the stored batch differs from what was signed: the strongest tampering signal there is.
`signature does not verify under kid "…"`	The signature is not over this `hash` with this key.	`hash` or `signature` was altered, or the JWK Set is not the platform's. Re-fetch the keys yourself first.
`kid "…" is not in the JWKS`	The page names a key the JWK Set does not publish.	A stale or foreign JWK Set, or, on a dedicated deployment, a signing key that was changed: compare with what your key-management team handed over.
`alg is "…", want "ES256-DER"`	An algorithm the platform never uses.	The record was altered.
`chain_hash recomputes to …, batch records …`	The link to the previous batch is broken.	A batch was inserted, removed or replaced before this one, or the wrong `-prev` was passed for the first batch of the page.
`sequence N follows M`	A gap or reordering.	A batch is missing from the page or the page was edited.
`chain link to the previous batch not checked: pass -prev`	Not a failure: the page does not start at sequence 1 and no previous head was given.	Chapter 4, step 4.

Step 2: Break the page on purpose

A verifier that has never failed proves nothing. Two alterations, each on a copy:

```
jq '.items[0].data[0].eventType |= sub("indykite";"indykyte")' response-
logs.json > tampered-data.json   # 1: one character changed inside an event
go run audit-verify.go -logs tampered-data.json -jwks response-jwks.json; echo
"exit=$?"
jq 'del(.items[1])' response-logs.json > tampered-gap.json
# 2: a batch dropped from the middle of the page
go run audit-verify.go -logs tampered-gap.json -jwks response-jwks.json; echo
"exit=$?"
```

The first copy fails its first batch on `data hashes to …`: the hash no longer matches, and nothing else, since the signature was never over the altered bytes but over the recorded hash, which is intact. The second copy fails on batch 3 with both `sequence 3 follows 1` and

`chain_hash recomputes to …`: its link was computed from batch 2's head, which is no longer on the page. Both exit with `1`. Delete the copies afterwards; keep only what was served.

Step 3: After the test passes

- **Run the full export once.** This page proves the batches; the **audit trail tutorial** exports manifests and checkpoints as well and verifies the chain end to end with **audit-export.sh and audit-verify.py**. A checkpoint is what later proves the chain was not quietly truncated, so an acceptance without one is incomplete.
- **Ask for the integrity verifier's exit code** on a dedicated deployment. The platform's own scheduled verifier reads the stored chains directly: `0` is clean, `2` means findings, any other value means the job failed to run. Your page check and its run should agree; if yours passes and it reports findings, the stored chain differs from what the API serves, which is exactly the case it exists for. The **infrastructure guide** lists the alerts to put on it and on the dead-letter queue.
- **Then take the one-way decision.** Lock the write-once bucket's retention (GCP) or switch it to `COMPLIANCE` mode (AWS) only now, with this test and the full export as the evidence that the pipeline works in that environment. Before the lock, a project owner could still remove the policy; after it, nobody can.
- **Repeat on a schedule.** The two curl calls and the `go run` are a four-line job. Keep every page and JWK Set it fetches: a chain verified today and a chain verified next month, together, prove that nothing between the two was rewritten.