How a sealed audit chain works

Four synthetic screening records, sealed with Eagle's linear-chain method, with hashes you can recompute yourself.

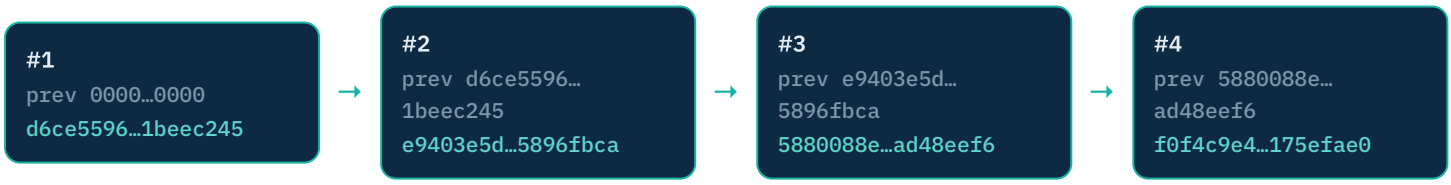
In brief

Every interactive screening in Eagle writes an audit record in the same database transaction as the decision (batch runs are recorded as one batch entry). Each record is turned into one exact string of bytes, then sealed with a keyed hash (HMAC-SHA256) over the **previous record's hash** plus its own bytes. Changing any field of any record changes its hash and breaks every link after it. Database triggers also reject updates and deletes (apart from Eagle's retention process for expired personal data), and an integrity check can recompute the whole log on demand.

Eagle seals its log as a Merkle tree with keyed checkpoints, locked on by default; the linear chain shown here is its configurable fallback, used in this sample because it is the easiest to recompute by hand.

The chain

SEQ	SEARCHED NAME	OUTCOME	TIME (UTC)	PREVIOUS HASH	ENTRY HASH
1	Example Supplies Ltd	PASS	08:14:02	00000000…00000000	d6ce5596…1beec245
2	Sample Person A	REVIEW	08:15:40	d6ce5596…1beec245	e9403e5d…5896fbca
3	Sample Person B	PASS	08:17:11	e9403e5d…5896fbca	5880088e…ad48eef6
4	Example Trading Co	REVIEW	08:19:55	5880088e…ad48eef6	f0f4c9e4…175efae0



Each record's "previous hash" is the entry hash of the record before it. The first record starts from 32 zero bytes.

What a record contains

The searched name, the number of results, the outcome (PASS, REVIEW or FAIL), who ran it and for which tenant, the version label of the sanctions data in force, a request ID, the time, and details including how fresh the lists were at the moment of screening. Page 2 shows every field of all four records.

Step 1: the canonical bytes

Before hashing, a record is written as fields in a fixed order, each as `key`US`value`, joined by RS (the ASCII unit and record separators, shown here as symbols). Values are JSON-encoded; nested details use sorted keys. Record #2 becomes:

```
v␟2␞
seq␟"2"␞
searchedName␟"Sample Person A"␞
resultCount␟3␞
ipAddress␟"10.20.0.14"␞
actorUserId␟"analyst-demo-01"␞
sanctionsVersionId␟"INGEST-2026-09-25T04:00Z-OFAC_SDN+UN+EU_FROZEN+UK_HMT"␞
tenantId␟"demo-bank"␞
outcome␟"REVIEW"␞
requestId␟"req-0002-demo"␞
details␟"{\"assurance\":\"screening-v2\",\"listCount\":4,\"oldestListAgeHours\":4.2}"␞
timestamp␟"2026-09-25T08:15:40.502Z"
```

Step 2: the keyed hash

`entry_hash = HMAC-SHA256( key, previous_hash ‖ canonical_bytes )`

Using a key means that someone with database access, who knows the algorithm, still cannot forge a valid chain without a secret the database never holds. In this sample the key is a published demo value, `"saolix-demo-key-not-a-secret"`, so you can check the hashes yourself.

All four records and their hashes

Every field value that was hashed, followed by the record's previous hash and entry hash, so any record can be recomputed.

```
{"seq": 1, "searchedName": "Example Supplies Ltd", "resultCount": 0, "ipAddress": "10.20.0.14", "actorUserId": "analyst-demo-01",
"sanctionsVersionId": "INGEST-2026-09-25T04:00Z-OFAC_SDN+UN+EU_FROZEN+UK_HMT", "tenantId": "demo-bank", "outcome": "PASS", "request
Id": "req-0001-demo", "timestamp": "2026-09-25T08:14:02.118Z", "details": {"assurance": "screening-v2", "listCount": 4, "oldestList
AgeHours": 4.2}}
  prev 0000000000000000000000000000000000000000000000000000000000000000
  hash d6ce5596b9ed04b8627ff18594144089e196e7cf604ee8074051f7ec1beec245
{"seq": 2, "searchedName": "Sample Person A", "resultCount": 3, "ipAddress": "10.20.0.14", "actorUserId": "analyst-demo-01", "sanct
ionsVersionId": "INGEST-2026-09-25T04:00Z-OFAC_SDN+UN+EU_FROZEN+UK_HMT", "tenantId": "demo-bank", "outcome": "REVIEW", "requestId":
"req-0002-demo", "timestamp": "2026-09-25T08:15:40.502Z", "details": {"assurance": "screening-v2", "listCount": 4, "oldestListAgeHo
urs": 4.2}}
  prev d6ce5596b9ed04b8627ff18594144089e196e7cf604ee8074051f7ec1beec245
  hash e9403e5dd052fbc9a79d7cc3d124e16b850f4cc433d579300baa40c35896fbca
{"seq": 3, "searchedName": "Sample Person B", "resultCount": 1, "ipAddress": "10.20.0.14", "actorUserId": "analyst-demo-01", "sanct
ionsVersionId": "INGEST-2026-09-25T04:00Z-OFAC_SDN+UN+EU_FROZEN+UK_HMT", "tenantId": "demo-bank", "outcome": "PASS", "requestId":
"req-0003-demo", "timestamp": "2026-09-25T08:17:11.930Z", "details": {"assurance": "screening-v2", "listCount": 4, "oldestListAgeHo
urs": 4.3}}
  prev e9403e5dd052fbc9a79d7cc3d124e16b850f4cc433d579300baa40c35896fbca
  hash 5880088e4762e49995886739f2a7a5d047bd00597a09987315a78c81ad48eef6
{"seq": 4, "searchedName": "Example Trading Co", "resultCount": 5, "ipAddress": "10.20.0.14", "actorUserId": "analyst-demo-01", "sa
nctionsVersionId": "INGEST-2026-09-25T04:00Z-OFAC_SDN+UN+EU_FROZEN+UK_HMT", "tenantId": "demo-bank", "outcome": "REVIEW", "requestI
d": "req-0004-demo", "timestamp": "2026-09-25T08:19:55.064Z", "details": {"assurance": "screening-v2", "listCount": 4, "oldestListA
geHours": 4.3}}
  prev 5880088e4762e49995886739f2a7a5d047bd00597a09987315a78c81ad48eef6
  hash f0f4c9e4db72937d638c5bf513583938a025d14bc7e52c1278e9a253175efae0
```

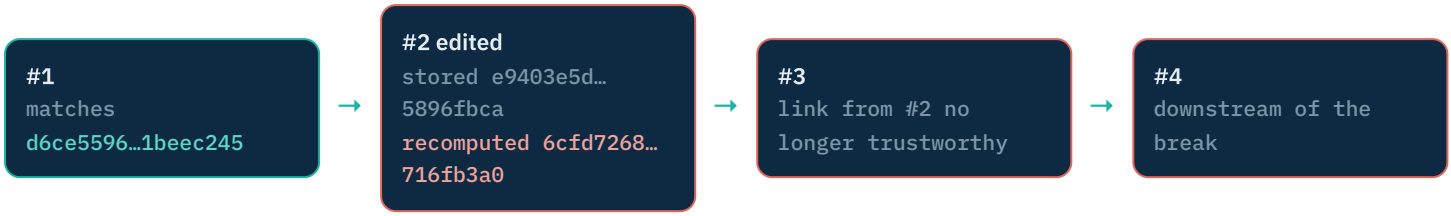
Step 3: verify

An integrity check recomputes each hash from the stored record and compares it with what was stored:

```
{"verified": true, "checked": 4, "firstBreak": null}
```

What tampering looks like

Suppose someone later edits record #2, changing its outcome from **REVIEW** to **PASS** so a flagged name looks clean. They leave the stored hashes untouched.



The recomputed hash of #2 no longer matches the stored one, so the check stops there and reports exactly where the chain broke:

```
{"verified": false, "checked": 2, "firstBreak": {"seq": "2", "expected":
"6cfd7268fd13347725cd730c777d1a65e9ada17fa95ebf87d62e9432716fb3a0", "actual":
"e9403e5dd052fbc9a79d7cc3d124e16b850f4cc433d579300baa40c35896fbca"}}
```

To hide the edit, they would have to recompute #2 and every record after it, which requires the key. In practice they would also be stopped earlier: Eagle's database triggers reject updates and deletes on the audit tables (the only exception is Eagle's retention process, which can erase personal data from expired records and leaves a marker).

Recompute it yourself

This Python reproduces Eagle's canonical form and hashing for chain version 2. Feed it the records above, starting from 32 zero bytes, and you will get the hashes on page 2.

```
import hashlib, hmac, json
KEY = b"saolix-demo-key-not-a-secret"
def stable(v):
    if v is None: return ""
    if isinstance(v, dict):
        return "{" + ",".join(json.dumps(k) + ":" + stable(v[k]) for k in sorted(v)) + "}"
    return json.dumps(v)
def canonical(r):
    f = [("v", 2), ("seq", str(r["seq"])), ("searchedName", r["searchedName"]),
         ("resultCount", r["resultCount"]), ("ipAddress", r["ipAddress"]),
         ("actorUserId", r["actorUserId"]), ("sanctionsVersionId", r["sanctionsVersionId"]),
         ("tenantId", r["tenantId"]), ("outcome", r["outcome"]), ("requestId", r["requestId"]),
         ("details", stable(r["details"])), ("timestamp", r["timestamp"])]
    return "\x1e".join(k + "\x1f" + stable(v) for k, v in f)
def entry_hash(prev, r):
    return hmac.new(KEY, prev + canonical(r).encode(), hashlib.sha256).digest()
```

What this proves, and what it does not

A verified chain proves the audit records have not been altered since they were written, and that none were removed from the middle. It is evidence about the record. For how Eagle reaches the decision in the first place, see the Insights article on the determinism boundary at saolixgroup.com/insights.