GUIDE

How to make AI agent audit logs tamper-evident: hash-chained OPA decision logs with external checkpoints

Chain the records. When your log collector receives OPA decision log events, give each record the SHA-256 hash of the record before it. Editing, deleting or inserting a record then breaks the chain at that point.

A chain cannot show that its newest records were cut off, so also publish a signed checkpoint of the latest hash at fixed times, to a place the log writer cannot change. An auditor can then recompute the chain and compare it with the checkpoints.

What "tamper-evident" means here

It means any change is detectable. It does not mean changes are impossible. You get detectability from the hash chain and the checkpoints, and you make changes hard with write-once storage and separate access.

Where each step runs

Step	Where	Notes
Decide and log	Your OPA	Decision logs carry `decision_id`, `input`, `result` and the bundle revision. Mask sensitive fields with `system.log.mask` first. (OPA decision logs)
Chain	Your log collector	OPA does not chain logs itself. The collector adds `prev_hash` and `hash` to each event in arrival order.
Checkpoint	A separate system	Sign the latest hash every few minutes and store it with a different owner or a public timestamp service.
Store	Write-once storage	Keep records where the writer cannot delete them during the retention period.
Verify	Auditor or scheduled job	Recompute the chain and compare it with the checkpoints.

Certificate Transparency logs use Merkle trees for the same goal at larger scale, with proofs that one log version extends another. (RFC 9162) A simple chain is enough for most decision logs. Move to a Merkle tree when you need proofs for single records without handing over the whole log.

Example: chain and verify

Run in Python 3. Five tests show that edits, deletes and inserts are detected, and that cutting off the newest records is detected only by comparing with a checkpoint.

```python
"""Hash-chain OPA decision log events so any edit, insert or delete is detectable.

Run in your log collector, after OPA's decision log plugin delivers a batch.
Each record stores the SHA-256 of the previous record. Keep the latest hash
somewhere the log writer cannot change (for example a signed daily checkpoint).
"""
import hashlib
import json

GENESIS = "0" * 64

def canonical(event: dict) -> bytes:
    return json.dumps(event, sort_keys=True, separators=(",", ":")).encode()

def chain(events: list[dict], prev: str = GENESIS) -> list[dict]:
    out = []
    for e in events:
        h = hashlib.sha256(prev.encode() + canonical(e)).hexdigest()
        out.append({"event": e, "prev_hash": prev, "hash": h})
        prev = h
    return out

def verify(records: list[dict], prev: str = GENESIS) -> int | None:
    """Return the index of the first bad record, or None if the chain holds."""
    for i, r in enumerate(records):
        expected = hashlib.sha256(prev.encode() + canonical(r["event"])).hexdigest()
        if r["prev_hash"] != prev or r["hash"] != expected:
            return i
        prev = r["hash"]
    return None
```

What to hand an auditor

1. The records for the period, with `prev_hash` and `hash`.
2. The signed checkpoints for the same period, and the public key.
3. The verifier script.
4. The bundle revisions named in the records, linked to the reviewed policy commits.
5. For approved high-impact actions: the signed approval tokens (or their IDs and digests), the approval service's public keys for the period, and the approver list in force.

Signed human approval as evidence

When an agent's high-impact action needed a person's OK, the decision record has to show that the approval was real. An approval that is just a row in a database, or "approved: true" in a request, only proves someone wrote that value. A signed approval proves who approved, what exactly they approved, and when it expires. Anyone holding the public key can check it later.

What a signed approval should bind:

Claim	Why it matters
Approver (`sub`)	Who approved. Must not be the requester
Requester (`req`)	The user the agent acted for
Digest of the exact call (`dig`): SHA-256 of tool and arguments	An approval for a $900 refund can't be replayed for $9,000
Audience (`aud`): the tool	An approval for one tool can't be used on another
Expiry (`exp`) and ID (`jti`)	Short-lived and traceable
Issuer (`iss`) and signature	Only your approval service can create one

How OPA uses it. The agent or gateway passes the approval token with the call. OPA verifies it with `io.jwt.decode_verify`, which checks the signature, expiry, issuer and audience. (OPA token built-ins) The policy then recomputes the digest of the call and compares it with the approved one. The approval ID, the approver and the digest go into the decision result, so they appear in the decision log. That log is chained and checkpointed as described above.

Tested with OPA v1.22.4 (10 `opa test` cases): a valid approval allows the call and records the evidence. A forged signature, an expired approval, one replayed for a larger amount, one used on another tool, a self-approval and an approver not on the list are each denied with a specific reason. The test signs tokens with a throwaway key.

REGO

```rego
package agent.approval

import rego.v1

# High-impact agent actions need a signed approval from a person other than the requester.
# The approval service signs a short-lived JWT (ES256) with these claims:
#   iss: "approvals.example.com"   aud: the tool name
#   sub: approver id               req: requester (the user the agent acts for)
#   dig: sha256 of {"tool", "args"} for the exact call   jti: approval id   exp: expiry
# OPA verifies the signature and every binding, then records the approval in the decision log.

high_impact := {"issue_refund", "delete_repository", "wire_transfer"}

approvers := {"u-bob", "u-carol"}

default allow := false

allow if {
          not input.tool in high_impact
          count(deny) == 0
}

allow if {
          input.tool in high_impact
          count(deny) == 0
          approval_ok
}

deny contains "missing user identity" if not is_string(object.get(input, "user", null))

call_digest := crypto.sha256(json.marshal({"tool": input.tool, "args": input.args}))

# io.jwt.decode_verify checks the signature, exp/nbf against "time", iss and aud.
verified := [valid, header, claims] if {
          [valid, header, claims] := io.jwt.decode_verify(input.approval_token, {
                    "cert": json.marshal(data.approval_keys),
                    "iss": "approvals.example.com",
                    "aud": input.tool,
                    "time": time.now_ns(),
          })
}

claims := verified[2] if verified[0]

approval_ok if {
          claims.dig == call_digest
          claims.req == input.user
```

```
            claims.sub != input.user
            claims.sub in approvers
}

# Why a high-impact call was not allowed, for the decision log and the agent.
approval_problem := "no approval token" if {
            input.tool in high_impact
            not input.approval_token
} else := "signature, issuer, audience or expiry check failed" if {
            input.tool in high_impact
            not claims
} else := "approval is for different arguments" if {
            input.tool in high_impact
            claims.dig != call_digest
} else := "approver may not approve their own request" if {
            input.tool in high_impact
            claims.sub == input.user
} else := "approver is not on the approver list" if {
            input.tool in high_impact
            not claims.sub in approvers
}

# Evidence fields to keep with the decision (they land in OPA's decision log result).
evidence := {
            "approval_id": claims.jti,
            "approver": claims.sub,
            "call_digest": call_digest,
            "approval_expires": claims.exp,
} if approval_ok

decision := {"allow": allow, "evidence": evidence} if approval_ok

else := {"allow": allow, "problem": approval_problem} if approval_problem

else := {"allow": allow}
```

Two details that decide whether this holds up:

1. **Compute the digest one way.** If the approval service and OPA serialize the arguments differently, the digests won't match. The simplest fix is to let OPA compute the digest and return it when approval is needed. The approval service then signs that value. If the digest has to be computed outside OPA, use a canonical JSON form such as RFC 8785.
2. **Keep the signing key away from the agent.** If the agent's runtime can reach the approval key, it can approve itself. Keep the key in the approval service, and rotate it with a new `kid`. Publish old public keys for as long as you keep the logs.

Verified identity for the approver. Where an approval must show that a specific, identity-checked person stood behind it, the approval can carry a verifiable credential presentation. OpenID for Verifiable Presentations 1.0 is a final OpenID Foundation specification for presenting credentials. The W3C Digital Credentials API (a Working Draft) lets a browser request them. Record the verification result and the credential's issuer in the approval, not the raw credential.

Where EnforceAuth fits

EnforceAuth is the OPA/Rego authorization control plane, in the category we call Authorization Context Intelligence. Your own OPA runtimes make each runtime decision. EnforceAuth does not make that decision. You author, test and ship the Rego and bundles in Writ. Engines send decisions to Writ, and Writ feeds EvidenceIQ, an EnforceAuth capability that keeps a durable, searchable record of authorization decisions. EvidenceIQ records decisions and never makes or changes them. Herald MCP, EnforceAuth's hosted MCP server, lets you operate EnforceAuth from Claude, Cursor or ChatGPT. Your own OPA makes each runtime decision. Free tier: the first 1 million decisions a month are free, then $89 per additional million. (llms.txt, Pricing)

EnforceAuth deploys in the cloud region you choose, or self-hosted.

FAQ

How do I make AI agent audit logs tamper-evident?

Hash-chain the decision records so each one includes the hash of the one before, and publish signed checkpoints of the latest hash to a system the log writer cannot change. Edits, deletions and insertions break the chain, and comparing with checkpoints catches records cut from the end.

Does OPA sign or hash-chain decision logs?

No. OPA can verify signed policy bundles, and it records the bundle revision in each decision log event, but chaining or signing the log records is done in your log collector.

Is a hash chain enough to prove no records were deleted?

Not on its own. A chain detects changes inside it, but not records removed from the end. Signed checkpoints of the latest hash, stored separately, close that gap.

What is the difference between tamper-evident and tamper-proof logs?

Tamper-evident logs make any change detectable. Tamper-proof would mean changes are impossible. In practice you combine tamper-evident records with write-once storage and separate access.

How do I prove a human approved an AI agent's action?

Use a signed approval, not a flag. The approval service signs a short-lived token that names the approver, the requester, the tool, and a SHA-256 digest of the exact arguments. The policy engine verifies the signature and recomputes the digest before allowing the call, and the approval ID and digest go into the decision log.

Can an approval for one AI agent action be reused for another?

Not if the approval is bound to the call. Put a digest of the tool and arguments, the tool as audience, and a short expiry in the signed approval. A different amount, tool or time then fails the check.

Where should the approval signing key live?

In the approval service only, never where the agent runs. If the agent can reach the key, it can approve its own actions. Rotate keys with a new key ID, and keep old public keys for as long as you keep the logs.