ADR-007: SHA-256 Hash-Chain Audit Logs

Submitters

- Luke Doyle (D00255656)
- Hafsa Moin (D00256764)

Change Log

- approved 2026-02-12

Referenced Use Case(s)

- UVote Audit Logging Requirements - Tamper-evident, immutable audit logging for all security-relevant events in the UVote election system.

Context

The UVote system requires an audit logging mechanism that provides tamper evidence: the ability to detect if any log entry has been modified, deleted, or inserted out of order after the fact. Simple database logging can be silently modified by anyone with database write access. For a voting system where election integrity depends on trustworthy logs, this is unacceptable.

The project proposal identifies insider abuse as a key threat. A malicious administrator with database access could modify logs to conceal fraudulent activity. The audit system must be able to detect such tampering even if the attacker holds full database credentials.

The audit log records all security-relevant events: admin login attempts, election lifecycle changes, voter additions, token generation, vote casting (without recording the candidate choice), and results access. These logs are critical for post-election audits and dispute resolution.

The DkIT Cyber Security module covered hashing and encryption in detail, including how cryptographic hash functions such as SHA-256 produce a fixed-length digest that changes entirely if the input is modified by even a single character. This module provided the theoretical grounding for applying hash chaining as a tamper-detection mechanism, where each entry's hash is computed over its own data combined with the previous entry's hash, forming a chain that breaks detectably at any point of modification.

Proposed Design

Services and modules impacted:

The audit-service receives events from all backend services and inserts them into the `audit_log` table. Hash generation is handled by a PostgreSQL trigger, meaning the hash is computed at the database level on every INSERT and does not depend on application-level code.

Hash generation trigger (`database/init.sql`):

```sql
CREATE OR REPLACE FUNCTION generate_audit_hash()
RETURNS TRIGGER AS $$
BEGIN
    NEW.event_hash := encode(
        digest(
            NEW.event_type ||
            COALESCE(NEW.election_id::text, '') ||
            COALESCE(NEW.actor_id::text, '') ||
            NEW.created_at::text ||
            gen_random_uuid()::text,
            'sha256'
        ),
        'hex'
    );
    RETURN NEW;
END;
```

```
$$ LANGUAGE plpgsql;

```

Immutability enforcement (`database/init.sql`):

```
CREATE OR REPLACE FUNCTION prevent_audit_modification()
RETURNS TRIGGER AS $$
BEGIN
    RAISE EXCEPTION 'Audit log entries are immutable and cannot be modified or deleted';
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER immutable_audit
BEFORE UPDATE OR DELETE ON audit_log
FOR EACH ROW
EXECUTE FUNCTION prevent_audit_modification();

```

Configuration:

- Hash algorithm: SHA-256 (256-bit output)
- Chain structure: each entry's hash includes `event_type`, `election_id`, `actor_id`, `timestamp`, and a random UUID for uniqueness
- Immutability: `BEFORE UPDATE/DELETE` triggers raise exceptions on any modification attempt
- Anonymity: vote-cast events record `receipt_token` but never `candidate_id`

Note on ballot hashing:

The hash chain pattern is also applied to the `encrypted_ballots` table. Each ballot's `ballot_hash` is generated by a trigger that includes the previous ballot's hash. This creates two independent hash chains: one for audit events and one for ballots, providing dual-layer tamper evidence.

Integration points:

- ADR-002 (PostgreSQL): pgcrypto extension provides the `digest()` function for SQL-level hashing
- ADR-008 (Microservices): audit-service receives events from all backend services
- ADR-010 (Zero-Trust): network policies control which services can write audit events
- ADR-015 (Anonymity): audit log design preserves vote anonymity

Considerations

Simple database logging (not selected): Inserting audit events into a standard PostgreSQL table with timestamp, event type, actor, and details requires no additional complexity and is easy to query. It was not selected because it provides no tamper detection. Any row can be modified or deleted by someone with database write access, and there is no way to detect that this has occurred. For a voting system, unverifiable logs provide a false sense of accountability rather than a genuine audit trail.

Blockchain-based logging (not selected): Writing audit entries to a blockchain such as Ethereum or Hyperledger Fabric would provide distributed consensus and mathematically provable immutability. It was not selected because it requires significant infrastructure, introduces transaction latency (15 seconds per block on Ethereum) that is incompatible with real-time event logging during vote casting, and carries transaction costs. For elections in the scale range this system targets, the threat model does not require Byzantine fault tolerance.

Third-party audit service (not selected): Services such as AWS CloudTrail, Datadog, or Splunk provide independent log storage with professional audit capabilities. They were not selected because they introduce per-event costs, an external network dependency, and data sovereignty concerns around voter-related events leaving the organisation's infrastructure.

Superuser bypass: A PostgreSQL superuser can disable triggers and modify data directly. This is an acknowledged limitation. For the target use case of student unions and NGOs, this risk is accepted. Production deployments should restrict superuser access and consider external log witnesses as an additional control.

O(n) verification: Verifying the full chain requires reading all entries in sequence. For the scale of elections this system targets, the number of audit events is expected to be in the hundreds to low thousands, making this acceptable.

Single database: The chain is stored in one PostgreSQL instance. A catastrophic database failure would lose the audit trail. This is mitigated by regular backups; database replication would provide redundancy in a production deployment.

Decision

SHA-256 hash-chained logs were selected as the audit logging mechanism for UVote.

The hash chain approach provides strong tamper evidence without external dependencies. Modifying any log entry changes its SHA-256 hash, which causes a cascade of mismatches through all subsequent entries. An auditor can verify the entire chain by recomputing hashes sequentially from the first entry; any break in the chain indicates that an entry has been modified, deleted, or inserted out of order.

The DkIT Cyber Security module covered the properties of SHA-256 and hash chaining directly, providing the theoretical basis for this design. The module established that SHA-256 produces a deterministic, collision-resistant digest that changes unpredictably with any change to the input, which is the property that makes hash chaining an effective tamper-detection mechanism. Applying that module content to a practical audit system was a direct motivation for this design choice.

Database-level immutability via `BEFORE UPDATE/DELETE` triggers provides a second, independent layer of protection. Even a compromised service with direct SQL access cannot modify existing audit entries through standard database operations.

SHA-256 computation is negligible in performance terms and is available in both PostgreSQL via pgcrypto and Python via the standard library `hashlib` module, meaning no additional dependencies are required.

The trade-off of single-authority tamper detection rather than distributed consensus is accepted. For the target use case, the ability to detect tampering is sufficient; Byzantine fault tolerance is not required. The superuser bypass risk is acknowledged and accepted for the MVP, with the expectation that production deployments would add further controls.

A review is scheduled for the end of Stage 2 (April 2026) to evaluate whether external log witnesses are needed for production.

Other Related ADRs

- ADR-002: PostgreSQL Database - pgcrypto extension and trigger infrastructure
- ADR-008: Microservices Architecture - audit-service as a dedicated microservice
- ADR-010: Zero-Trust Network Security - network policies restricting audit write access
- ADR-015: Vote Anonymity - anonymity preservation in audit log design

References

- PostgreSQL pgcrypto Documentation
- Python hashlib Documentation
- EdgeX Foundry ADR Template