

​​



**TAZAMA SECURITY DESIGN DOCUMENT**

**​​Business Requirement Specification Document​**

**1. Introduction**

This document outlines the security design of the Tazama platform, demonstrating how authentication, authorization, data protection, and monitoring are implemented across its key components. The design includes:

* **Minimum Mandatory Requirements** that Tazama already have in place (or must have at a baseline), ensuring essential security measures are consistently applied.
* **Additional Improvement Recommendations** that go beyond the current design, aimed at strengthening Tazama’s future security posture.

Information and decisions here were developed from a combination of workshops, technical Q&A, and analysis of Tazama’s architecture diagrams. The intent is to provide a clear overview of how Tazama protects transaction data at scale, identifies suspicious activity, and enforces secure access to its APIs.

**2. Scope of the Security Design Document**

**2.1 In Scope**

This security design addresses the **application-layer** elements of Tazama, specifically:

* **Identity and Access Management (IAM)**: Issuing and validating tokens, managing user and service credentials, and delegating authorization.
* **Data Security**: Encompassing all measures that protect confidentiality, integrity, and availability of data—such as role-based access, database authentication, and encryption.
* **Platform Logging, Auditing, and Monitoring**: Capturing operational and security-related events, providing an audit trail of critical actions, and monitoring to detect potential threats.

This design focuses solely on how Tazama implements application-level security across its microservices, databases, and Identity and Access Management (IAM) components.

**2.2 Out of Scope**

Network and infrastructure controls (firewalls, Kubernetes cluster security) or operational security aspects (CI/CD pipeline hardening, incident response policies) are out of the scope of this security design. They will be addressed in infrastructure‐focused or operational‐focused documents as part of the Sandbox and also discussed in client implementations on a case-by-case basis.

**2.3 Deferred from Scope**

The security design for the Neural Autonomic Transport System has been deferred from the current scope and will be at later stage as an addendum to this document.

**3. IAM (Identity and Access Management)**

Tazama uses Keycloak as its external IdP, integrated via an Auth‐service and a shared Auth‐lib. Users and services obtain a “tazamaToken” (JWT) by authenticating against Keycloak, and all Tazama APIs require a valid token. The following subsections detail how API authentication, user management, and service-to-service authentication are structured.

CRITICAL: All API authentication mechanisms described herein *must* be explicitly enabled and configured as active in production or any secure environment. Development or demonstration configurations that bypass authentication must not be used in production

**3.1 API Authentication**

**Minimum Mandatory Controls**

1. Token-Based Authentication for All APIs:
   1. Tokens are initially issued by the IAM provider (Keycloak). Tazama repackages the Keycloak JWT as a more generic Tazama JWT token.
   2. Every Tazama endpoint (TMS, Admin Service) requires a valid JWT (tazamaToken).
   3. To obtain this token, clients must authenticate with Keycloak (via the Auth‐service).
2. Integration with Keycloak (OIDC): The Auth‐service acts as a confidential client to Keycloak, exchanging user credentials for a Keycloak token, then transforming that token into Tazama’s own JWT format for consistency across Tazama microservices.
3. Cryptographic Signing of Tokens: Tazama issues tokens signed by an internal private key; each Tazama service verifies the signature with the corresponding public key. Any alteration to the token is thus detected and rejected.
4. Central Auth‐lib for Validation: All Tazama services rely on the same Auth‐lib function (validateTokenAndClaims()) to parse, verify, and check claims, ensuring uniform access control rules.
5. Defined Token Lifetimes: Tazama enforces expiration on tokens to limit their valid window. For system‐to‐system flows (common in Tazama’s real‐time environment), tokens can be slightly longer‐lived to reduce overhead.
6. Tokens Include Embedded Claims: All tokens must include embedded claims defining user roles and scopes that control access to various services and functions.

**Additional Improvement Recommendations**

1. End‐to‐End TLS: Confirm that the login flow (client to Auth‐service, Auth‐service to Keycloak) is always encrypted with TLS, validating certificates to avoid man‐in‐the‐middle attacks.
2. Secrets Management: Implement a secret vault solution to handle dynamic secrets, certificate issuance, and short-lived credentials.
3. Multi‐Factor Authentication (MFA): For admin or other high‐privilege accounts, enable Keycloak’s MFA features to harden login security.
4. Short‐Lived Tokens with Refresh Flows (for User Interactions): Introduce refresh tokens to keep access tokens short‐lived (e.g., 5–15 minutes). This reduces the risk of an access token being compromised.
5. Brute‐Force Attack Mitigation: Enable Keycloak’s built‐in detection for excessive failed logins and optionally rate‐limit at the Auth‐service’s /login endpoint.
6. Immediate Revocation Mechanism: Enhance Tazama to revoke or invalidate active tokens if an account is compromised (rather than waiting for expiration). This is critical in high‐security scenarios.
7. Claims for Contextual Binding: Enhance token security by including contextual binding attributes (e.g., originating IP address, device fingerprint, client application ID) in token claims. Services should then validate these claims against the current request context to mitigate token theft and replay attacks. Consideration should be given to the reliability of sourcing these attributes and potential impacts on user experience (e.g., legitimate IP changes).
8. Token Binding to Browser: Implement token binding to prevent token theft and reuse from different browsers

**3.2 User Management**

**Minimum Mandatory Controls**

1. Centralized User Directory: All user accounts (including administrators) are stored in Keycloak. Tazama does not keep passwords internally, delegating to Keycloak for credential storage and verification.
2. Roles Assigned in Keycloak: Roles (e.g., TMS\_Admin, TMS\_User) exist in Keycloak; these are included as claims in the JWT. Tazama’s services interpret these roles to permit or deny actions.
3. Management via Keycloak Console or API: Tazama operators use Keycloak’s admin UI or admin APIs to create, update, or remove users, set passwords, and assign roles.
4. Separation of Administrative Duties: The TMS and Admin Service are restricted to the necessary roles. Complete platform or realm‐wide user tasks (like creating new admin roles) remain inside Keycloak’s console.
5. Basic Lifecycle Processes: Administrators can revoke user credentials, reset passwords, or deactivate accounts via Keycloak. Tazama honors the updated claims once the user obtains a new token.

**Additional Improvement Recommendations**

1. Strengthened Password Policies: Enforce minimum length, complexity, and password history. Possibly set periodic expirations or require a reset after a specific time.
2. Regular Role and Access Reviews: Administrators should periodically check which users still need which roles, removing any excess privileges or stale accounts.
3. Log Keycloak Admin Events Centrally: Stream “Keycloak admin events” (changes to user permissions, realm settings) into Tazama’s main logging or SIEM for complete visibility.
4. Adaptive/Contextual Access: Consider forcing an MFA step for high‐risk actions or unusual geolocations.
5. Session Visibility: Provide administrators with a real-time session viewer to monitor and terminate active sessions when necessary.

**3.3 Service‐to‐Service Authentication**

**Minimum Mandatory Controls**

1. Auth‐Service to Keycloak: The Auth-Service uses a secure client secret (stored outside the codebase) to authenticate with Keycloak’s token endpoint. Keycloak only issues tokens to recognized clients.
2. User Token Propagation: If one Tazama microservice calls another “on behalf of a user,” it forwards the same user JWT so the callee can re‐check roles. This preserves the end‐user context throughout the microservice chain.
3. Private Network Boundaries: Internal Tazama components (Event Director, Rule Processors, etc.) communicate on a private cluster network. They typically do not expose endpoints to the public internet.
4. Optional Keycloak Service Clients: Tazama can create service‐oriented credentials in Keycloak for purely machine‐to‐machine flows, though the user context is often passed along.
5. **All Internal Service Calls Use JWT**: All service-to-service communication within Tazama must use JWT-based authentication

**Additional Improvement Recommendations**

1. Mutual TLS: Implement service‐to‐service TLS with each microservice presenting a unique certificate. This prevents unauthorized microservices (or rogue pods) from impersonating legitimate services.
2. Dedicated Service Account Tokens: Where a microservice acts without a user context, register a Keycloak client using a strictly limited scope or role.
3. Short-lived service Tokens: Service tokens should be short-lived and automatically rotated to limit the impact of compromised credentials.
4. Establish Non-Static Service-to-Service Trust Relationship: While services authenticate with JWTs, they are implicitly trusted once verified, and to mitigate against mutual trust abuse, consider the following:
   * Integrating mTLS and workload identity (e.g., SPIFFE/SPIRE) to validate microservice identity per call
   * Periodically rotate service identities and enforce identity verification (such as enclave status or container integrity) for sensitive flows.
5. Service Mesh Integration: Istio or Linkerd can automate TLS between pods, plus apply policies like “Service A can only call Service B’s endpoints if it has role X.”
6. Continuous Authorization or Real-Time Risk-Based Decisions: Ensure that once a token is issued, access decisions do not depend solely on static claims but utilize real-time contextual risk evaluation (e.g., IP anomalies, device trust) by either:
   * Incorporating risk-adaptive access controls like Keycloak extensions or policy engines like OPA/Gatekeeper.
   * Continuously re-evaluating trust using contextual data (IP reputation, geo, behaviour baseline).
7. OAuth2 Token Exchange: A token exchange flow can produce a narrower token if microservice B only needs a subset of the original user’s permissions. This reduces over‐privileged tokens in internal calls.
8. Credential Usage: Log when a service credential is used. If it appears from an unexpected IP or at an unusual rate, generate a security alert.

**4. Data Security**

Tazama processes sensitive financial transactions, so ensuring strong data protection is paramount—both at rest and in transit. The platform enforces role-based restrictions, unique credentials per service, encryption mechanisms, and minimal privileges to reduce the risk of data leakage or unauthorized modification.

**4.1 Role‐Based Access Control (RBAC)**

**Minimum Mandatory Controls**

1. Role‐to‐Privilege Mapping: Each Tazama endpoint or function checks for specific roles. For example, only users with an “Admin” role can alter rules or typologies.
2. Enforcement in Microservices: The Auth‐lib automatically denies requests unless the JWT includes the required role claim. This check runs on every request to TMS, Admin, or another service.
3. Default Deny Posture: New users have no roles, ensuring they cannot inadvertently access Tazama features without explicit assignment.
4. Admin/User Role Segregation: Administrative tasks (like modifying system config) are not included in standard user roles, preventing accidental overreach.
5. Documented Roles: The Tazama repository or documentation includes definitions of each role (e.g., TMS\_User, TMS\_Admin) and the associated access rights.

**Additional Improvement Recommendations**

1. Granular Role Expansion: As Tazama adds more modules or new features (e.g., a case management UI), ensure each major capability corresponds to a distinct role.
2. Segregation of Duties Enhancements: For high‐risk changes (like rules that can block transactions), separate those privileges so no single user can create and override without a second approval.
3. UI Role Awareness: If Tazama or external front‐ends are used, hide or turn off UI elements unless the user’s token has the relevant role. This reduces error rates and user confusion.
4. Attribute‐Based Access Control (ABAC): If Tazama needs to enforce fine‐grained constraints (e.g., transaction geography, time‐based restrictions), consider checking user or transaction attributes within the Auth‐lib.

**4.2 Least Privilege**

**Minimum Mandatory Controls**

1. Scoped Service Responsibilities: The TMS API handles transaction ingestion and data prep, the Rule Processor only runs its logic, etc. Each microservice has a narrowly defined role.
2. Separate Credentials for Each Service: The TMS API, Admin Service, and other components do not share DB credentials. If one is compromised, it does not grant universal data access.
3. Admin Function Isolation: Configuration tasks (like turning a typology on or off) reside in the Admin Service, not the TMS. This inherently prevents user‐level endpoints from making admin changes.
4. Minimal Data Retrieval: Rule processors pull only the historical data needed for their specific check, typically via DataCache or targeted DB queries. They do not fetch entire account sets.

**Additional Improvement Recommendations**

1. Centralized Secrets Management: Store DB passwords, encryption keys, and Keycloak client secrets in a vault solution, not just environment variables. This includes rotation policies and fine‐grained access.
2. Lock Down Container Privileges: Ensure Tazama containers run as non‐root, with read‐only file systems where possible, and no extra Linux capabilities.
3. Quarterly Permission Reviews: Review permissions quarterly and remove them when no longer needed.
4. API Route Review: Evaluate all API routes for over-broad access patterns
5. Continuous Permissions Review: Track which privileges each microservice used. If unneeded rights are discovered, remove them to reduce potential misuse.
6. Data Segmentation for New Modules: If Tazama expands to new DBs or data sets, keep them separate so a compromise in one service does not cascade across the entire data environment.

**4.3 Strong Database Authentication**

**Minimum Mandatory Controls**

1. Unique Database Accounts: Each service has a DB username/password. The application code does not use a “root” or “superuser” DB account, limiting damage from stolen credentials.
2. Secure Storage of Credentials: DB passwords and keys are never in source control. They are set at deployment time (e.g., via Kubernetes Secrets).
3. Configured Password Complexity and Rotation: Long, random passwords are used for DB accounts. If a breach occurs, credentials can be rotated by updating secrets and redeploying.
4. Authentication Enforced by ArangoDB: The DB does not accept unauthenticated connections. Each service must provide valid credentials.

**Additional Improvement Recommendations**

1. Network‐Level DB Restrictions: Only allow connections to ArangoDB from Tazama’s Kubernetes namespace or VPC. Deny external traffic to the DB port entirely.
2. Logging and Alerting on Failed DB Logins: Feed ArangoDB’s auth logs into the central monitoring system and create alerts if repeated failures occur.
3. Dynamic Password Issuance via Vault: Generate short‐lived DB credentials for each Tazama service, rotating them automatically. Stolen credentials then quickly become invalid.
4. Granular DB Roles: If a service only needs read access to specific collections, do not grant write or admin privileges. This enforces least privilege at the DB level.
5. Client Certificate Authentication: If supported, shift from password‐based DB auth to TLS certs, adding an extra layer of trust verification.

**4.4 Data at Rest**

**Minimum Mandatory Controls**

1. Database Encryption: ArangoDB’s data files and backups are encrypted (e.g., using disk encryption on the underlying host or ephemeral volumes in a cloud environment).
2. Encrypted Secrets in Cluster: Kubernetes’ etcd (where secrets might live) is configured with encryption enabled, preventing local reading of secrets from disk in plaintext.
3. Encrypted Backups: Tazama backups (database dumps, config exports) are stored encrypted, typically in a restricted S3 bucket or similarly secured location.
4. Minimal File System Permissions: Each microservice can only read/write the required directories. World‐readable or world‐writable files are avoided.

**Additional Improvement Recommendations**

1. Central Key Management: Use a cloud KMS or Vault to store and rotate encryption keys for ArangoDB volumes. This allows fine‐grained key policies and controlled re‐encryption.
2. Masking in Logs: Double‐check that logs never contain raw sensitive personal data in plaintext (e.g., account numbers, phone numbers). Only partial or hashed values should appear.
3. Field‐Level Encryption: Especially for sensitive fields (like personal IDs or specific transaction details), encrypt them at the application level before storing.
4. Regular Encryption Audits: Periodically verify that new data sets or extended features follow encryption policies. Confirm that backups are restorable and remain encrypted in transit and storage.

**4.5 Data in Transit**

**Minimum Mandatory Controls**

1. TLS for External Connections: The TMS API and Admin Service endpoints should be exposed only over secure HTTPS, with no HTTP fallback. Digital certificates used should be from authorized providers and validated to ensure trust (verify ownership).
2. Auth‐service <-> Keycloak Over HTTPS: User credentials or tokens are never sent in plaintext. Keycloak’s certificate is validated to prevent spoofing.
3. Internal Pod Security Assumptions: Within the Kubernetes cluster, Tazama often uses private networking for microservice communication (Event Director to Rule Processor, etc.).
4. Avoid Plaintext Sensitive Data: Even in internal communications or logs, Tazama ensures that especially sensitive values (e.g., user passwords, entire tokens, PII) are not transmitted in the clear or logged in the open.

**Additional Improvement Recommendations**

1. Mutual TLS Within the Cluster: Adopt a service mesh or an mTLS approach so each microservice verifies the other’s certificate. This reduces the risk if an attacker gains partial access to the cluster network.
2. Network Policy Enforcement: Use Kubernetes Network Policies to strictly limit which pods can talk to the DB or Keycloak, providing a second line of defense.
3. Strict Transport Security (HSTS): Enable HTTP Strict Transport Security on external endpoints to enforce HTTPS usage by browsers or other clients.
4. Automated Certificate Renewal: If exposing TMS or Admin endpoints externally, use cert-manager or Let’s Encrypt to manage certificate lifecycles so they never expire unexpectedly.
5. TLS Downgrade Checks: Periodically scan endpoints to ensure that only modern TLS versions (1.2/1.3) and strong cipher suites are accepted, removing outdated protocols.

**4.6 SQL Injection Protection**

**Minimum Mandatory Controls**

* Parameterized Queries: To prevent SQL injection attacks, all database queries must use parameterized statements. As a guideline:
  + Arango Query Language (AQL) provides for AQL-safe parameterization in TypeScript by using a template string generator function, e.g. <https://github.com/frmscoe/rule-054/blob/434a5fe040691dad84e7c8352edfa3e6a5121e17/src/rule-054.ts#L59>
* ORM Usage: Use ORM frameworks that automatically implement safe query practices.

**Additional Improvement Recommendations**

1. Static Code Analysis: Integrate static code analysis tools to detect potentially insecure query patterns.
2. Query Review: Regularly review query construction patterns, especially in code that handles dynamic filtering or sorting.

**4.7 Input Validation**

**Minimum Mandatory Controls**

1. API Boundary Validation: All user input must be validated at the API boundary before processing.
2. Type Checking: Implement strong type checking for all inputs to prevent type confusion attacks.

**Additional Improvement Recommendations**

1. OpenAPI Schemas: Use OpenAPI schemas for the structured validation of all API inputs.
2. Strong Typing: Implement strong typing throughout the codebase to prevent type-related vulnerabilities.
3. Input Sanitization: Sanitize inputs to prevent various injection attacks beyond just SQL injection.

**4.8 Separate Environments**

**Minimum Mandatory Controls**

1. Environment Separation: Development, test, and production environments must be logically and physically separated.
2. Controlled Data Flow: Control data flow between environments to prevent leakage of production data.

**Additional Improvement Recommendations**

1. Access Restrictions: Enforce strict access restrictions and data masking in non-production environments.
2. Production-Like Testing: Ensure non-production environments mirror production configurations to validate security controls.

**5. Platform Logging, Auditing, and Security Monitoring**

Tazama already produces operational logs for diagnosing issues, but logging and auditing also supply essential security intelligence. Each Tazama service can forward logs to a centralized store (e.g., ELK stack). Meanwhile, an audit trail ensures accountability for admin actions and access to sensitive data. Finally, integrated monitoring helps detect anomalies or potential intrusions in real time.

For fraud investigations, we are also recommending structured event logs with full transactional history to track user-driven changes to system documents at the field level, along with the need to record comments or reasons against certain high-risk updates.

**5.1 Activity Logging**

**Minimum Mandatory Controls**

1. Comprehensive Request Logging: The TMS API, Admin Service, and Auth‐service record each request with timestamps, endpoint details, user identity (when present), and success/failure results.
2. Centralized Aggregation: Logs are typically collected in Elasticsearch or a similar system, preventing data loss from container restarts. Operators can query or visualize all logs in one place.
3. Structured and Correlated Logs: Tazama uses correlation or trace IDs to link events across microservices (e.g., TMS -> Event Director -> Rule Processor). This makes debugging or forensic analysis more efficient.
4. Error/Exception Logging: JWT validation failures, DB connection errors, or rule processing exceptions are recorded with enough detail to diagnose problems (though without disclosing private keys or passwords).
5. Avoid Logging Sensitive Information: Tazama does not store full tokens, user passwords, or personal data in logs. Instead, partial identifiers or masked data are used where possible.

**Additional Improvement Recommendations**

1. Integrate Keycloak Logs: Incorporate Keycloak’s login and admin event logs into the same ELK environment to see the complete user authentication flow alongside Tazama actions.
2. Real‐Time Alerting on Suspicious Patterns: Configure watchers or alerts for repeated failed logins, unusual transaction spikes, or other triggers that might indicate an attack or misuse.
3. Extended Log Retention: Keep logs readily accessible for at least 90 days (hot storage), then archive them for one year or more if an incident is discovered later.
4. Privacy Guardrails: Re‐review what fields get logged if Tazama expands or changes data schemas. Where possible, store only event metadata and not entire personal data fields.
5. Link to APM Tools: If Tazama uses an Application Performance Monitoring solution, align logs with performance metrics to see security anomalies alongside performance anomalies.

**5.2 Audit Trails**

**Minimum Mandatory Controls**

1. Admin Action Logging: When administrators create new user accounts or modify roles (in the Admin Service or Keycloak), the event is logged. This ensures an auditable record of who changed what.
2. ArangoDB Write Accountability: Tazama logs DB writes in each service’s operational logs, so one can trace which service performed which write operation.
3. Immutable Log Storage: Once logs or audit records reach Elasticsearch or another aggregator, they cannot be altered or deleted without leaving a trace, preserving data integrity.

**Additional Improvement Recommendations**

1. Consolidate Keycloak Admin Events: Stream changes in Keycloak realm settings or user roles into Tazama’s central audit logs for unified visibility.
2. Alert on Critical Admin Ops: If a new high‐privilege user is created or certain rule thresholds are changed, generate an immediate alert to the security team.
3. Dedicated Audit‐Log Mechanism: Consider implementing an append‐only audit database or ledger for critical events, ensuring tamper‐evident properties.
4. Routine Audit Reviews: Security personnel should regularly review logs for suspicious admin actions (e.g., role escalations, significant batch user changes) or other anomalies.
5. Database Audit Plugin: If available in ArangoDB enterprise or other solutions, enable advanced read/write auditing on sensitive collections.

**5.3 Integrated Application Security Monitoring**

**Minimum Mandatory Controls**

1. Logging Sidecar & Dashboards: Tazama uses a sidecar pattern or direct log shipping to collect logs into Elasticsearch or a similar aggregator, giving real‐time operational visibility.
2. Rule & Typology Event Logging: Each rule or typology evaluation logs relevant details, enabling the correlation of suspicious patterns or repeated triggers across transactions.
3. Basic Performance Monitoring: CPU usage, memory, and request throughput are tracked to spot potential denial‐of‐service attempts or resource exhaustion.

**Additional Improvement Recommendations**

1. SIEM Integration: If the deploying organization has a broader Security Information and Event Management system, forward Tazama’s logs to it for correlation with network, host OS, or other logs.
2. User & Entity Behavior Analytics: Build baselines for normal behavior over time. If a user account or system suddenly performs far more queries or rule updates than usual, raise an alert.
3. Automated Incident Response Hooks: Scripted responses (deactivating an account, blocking traffic from an IP) can be triggered if certain suspicious thresholds are met.
4. Periodic Security Drills: Simulate an unauthorized login or brute‐force attempt to verify that Tazama’s real‐time monitoring and alerting systems respond effectively.
5. Continuous Tuning: Update detection rules and thresholds as Tazama’s environment changes, ensuring minimal false positives while still catching genuine threats.

**6. Appendix A - Visual Representation of High-Level Tazama Security Architecture**

**A diagram of security system

AI-generated content may be incorrect.**

1. **Keycloak + Auth‐service**:
   * **Authentication Flow**: A client or user system posts credentials to the Auth‐service, which verifies them with Keycloak.
   * **Token Generation**: Keycloak issues an OAuth token; the Auth‐service transforms it into Tazama’s JWT.
2. **Tazama Services** (TMS, Event Director, Rule Processors, Typology Processor, etc.):
   * **JWT Validation**: Each service uses the Auth‐lib to validate the token signature and claims for every request.
   * **Role‐Based Authorization**: Specific endpoints require specific roles.
3. **Data Security Layer**:
   * **ArangoDB**: Each microservice has separate DB credentials. Data is stored in an encrypted format at rest.
   * **Graph Model**: Tazama organizes transaction data in a graph. DB queries are done with minimal privileges.
4. **Monitoring and Logs**:
   * **Log Collection**: Tazama microservices and Keycloak can be piped into Elasticsearch.
   * **Alerts**: Real‐time watchers can detect unusual login attempts or suspicious transaction patterns.
   * **Audit Trails**: Administrative changes and suspicious event triggers are recorded for forensic analysis.

Below is the illustration of the 3-step authentication, authorization and logging process within the Tazama security architecture flow:

