Security Overview

MessageFoundry is a healthcare integration engine that carries PHI, so security is a first-class design constraint rather than an afterthought. This overview describes the controls that are built into the engine today: how it authenticates and authorizes every action, how it protects data at rest and in transit, how it produces a tamper-evident record of what happened, the development and CI practices behind the code, and the threat model for each interface. It closes with our OWASP ASVS 5.0 Level 3 self-assessment result, stated honestly — including what has and has not been independently reviewed.

This page is a summary. For the full engineering detail behind these controls, see the **Secure Development Standards** document (linked at the end). Where a control is opt-in or off by default, that is called out explicitly — we would rather be precise than impressive.

Secure by default

MessageFoundry is built to fail closed and to make the safe configuration the default one.

- **Authentication is required.** The running service attaches an authentication layer to every route except a `GET /health` liveness probe and a non-sensitive operational-config endpoint. The in-process embedding path is fail-closed: with no authentication service attached it denies every protected route until a developer explicitly opts out for local development.
- **Loopback by default.** The API binds to `127.0.0.1`. A non-loopback *plaintext* bind is refused at startup; exposing a listener beyond loopback requires TLS (in-process or terminated at a trusted upstream proxy). This is deployment guidance built into the product: bearer tokens and PHI never cross a network in the clear by accident.
- **Deny-by-default authorization.** Every protected route demands a specific permission. Anything not explicitly granted is denied.
- **First-run bootstrap is sealed.** On first start against an empty store, the engine creates a single bootstrap administrator with a random one-time password written to an owner-only file (never to the log). The account is flagged to force a password change on first sign-in, and it self-retires once a real second administrator exists (or after a configurable expiry if left unclaimed).

Identity and access control

Authentication

Operators authenticate as **local accounts** or against **Active Directory**, and every action is attributed to a distinct user in the audit trail — no shared logins.

- **Local accounts** use **argon2id** password hashing (tuned cost parameters, with a concurrency cap so a login flood cannot exhaust CPU). The password policy is aligned to OWASP ASVS 5.0: a 15-character minimum, no mandatory character-class composition (ASVS discourages it), offline breached- and common-password screening, a context-word deny-list, and rejection of passwords that contain the username. All screening is local — no third-party password API is called.
- **Active Directory accounts** authenticate by LDAP bind over **LDAPS** (TLS 1.2+, with certificate verification and nested-group resolution). AD security groups map automatically to engine roles — and to per-channel scope — and roles re-sync from the directory on every login.
- **Multi-factor authentication.** Local accounts use a native **RFC 6238 TOTP** second factor with single-use recovery codes; MFA is **required** for local accounts. The TOTP secret is stored encrypted at rest and recovery codes are argon2id-hashed. Enterprise and AD users receive MFA through their own identity provider via SSO federation, so a directory login is never prompted for an engine TOTP.

Authorization (RBAC)

Access control is **role-based and deny-by-default**. Roles are a fixed built-in set — Administrator, Operator, Deployment, Coding, Viewer, and Auditor — each mapping to permissions from a single catalog; holding multiple roles grants the union of their permissions. Every request re-resolves the user, roles, and permissions from server-side state (no caching), so a revoked privilege or a disabled account takes effect immediately.

Two finer-grained controls layer on top of route-level RBAC:

- **Per-channel scoping** confines an operator's message and connection access to a defined set of interfaces. Out-of-scope message reads return `404` (so they never reveal that a record exists), and denials are audited.
- **Field-level (per-property) authorization** gates individual PHI-bearing fields *within* a response, so a user can see an object without seeing its patient-identifying fields. The policy lives in one place and is applied to every returned row through a single helper, and a test fails the build if a new PHI-bearing field is added without being mapped — so the gate cannot be silently forgotten. Patient summaries (MRN, name) and free-text disposition fields require a summary-view permission; the raw message body requires a separate, coarser permission.

Defense-in-depth for sensitive actions

High-value and administrative operations are protected by several independent layers, not by network location alone:

- **Step-up re-verification** — sensitive admin operations require the session to have re-proved its credential (and satisfied MFA) within a short recent window, not merely to hold a valid token.
- **Dual-control approval** — high-impact operations (for example, bulk dead-letter replay and connection purge) can be configured to require a second, distinct approver before they execute; a requester can never approve their own request, and both identities are written to the audit log.
- **Contextual-risk step-up** — an optional signal forces a fresh step-up when a sensitive admin action arrives from a client address the session has not verified from.

Interface authentication

Beyond operator sign-in, the engine authenticates its **machine interfaces**:

- **mTLS** — mutual-TLS client-certificate authentication on transports that support it.
- **OAuth 2.0 client-credentials** — for REST/FHIR outbound calls.
- **SMART on FHIR Backend Services** — OAuth 2.0 `client_credentials` with a signed-JWT client assertion (RS384/ES384), opted in per connection; the engine mints a per-request bearer token and re-mints on a `401`, and the token endpoint is governed by the outbound egress allowlist. This is a client-only profile — there is no authorization-server facade.

Sessions

Sessions are **opaque server-side tokens** (not JWTs): the client holds the token, and the store keeps only its SHA-256 hash, so logout, expiry, and role changes take effect immediately. Each request enforces an **idle timeout** (default 30 minutes) and an **absolute lifetime** (default 12 hours). Session validation **fails closed on a backward wall-clock step** (for example an NTP step-back or a VM snapshot revert) rather than reviving an expired token, and the idle clock advances only on user-driven requests.

Changing a password, disabling a user, or an AD role/scope change on re-login revokes that user's sessions. A per-user concurrent-session cap applies (default 5). Users can list and revoke their own active sessions (including "sign out everywhere else"), and administrators can force-sign-out a user for offboarding or suspected compromise. Every revocation is audited. The desktop console stores its token in the OS keyring (Windows Credential Manager), sends it as a `Bearer` header, and refuses to send credentials over plaintext HTTP to a non-loopback host.

Data protection

Encryption at rest

Message bodies and the error/disposition columns are encrypted with **AES-256-GCM** (authenticated encryption), with a fingerprint-based **key-rotation keyring** (an active key plus retired decrypt-only keys) and a key-rotation command. On the default SQLite backend, rows that cannot be decrypted are routed to dead letters rather than crashing the service.

Two honest deployment notes apply. At-rest encryption **activates when an encryption key is set**; when PHI would otherwise be written in cleartext in a production or staging environment, the engine emits a startup warning, and it can be configured to refuse to serve without a key. Separately, the searchable summary column (MRN, patient name) is kept in plaintext **by design** so it can be indexed for search, and database write-ahead files hold recent data outside the application cipher. Closing that gap is the role of **operator-supplied full-volume encryption** (such as BitLocker or LUKS), which the engine documents as a deployment precondition but cannot enforce itself.

Redaction and logging

PHI is kept out of logs and exception text by construction. An exception-path redaction chokepoint scrubs HL7-shaped content from any value on its way to a log or a stored error column while preserving the exception type, and a global log filter redacts PHI and strips control characters (defending against log injection) on every record. Credentials, tokens, and message bodies are never logged. Debug-level logging is refused in a production environment.

Retention

An opt-in retention runner nulls message bodies while preserving metadata, then checkpoints and compacts the store, auditing each pass. Retention windows must be configured by the operator.

Tamper-evident audit

Every authentication and authorization event is written to a durable audit log with the acting user — sign-ins, failures, lockouts, permission denials, user and role changes, AD mapping changes, and PHI access (viewing a raw message or displaying patient summaries is recorded with the viewer). Credentials, tokens, and PHI bodies are never written to the audit trail.

Each audit row carries a `row_hash` that **chains the previous row's hash with this row's content (SHA-256)**, so deleting, editing, or reordering any row is detectable. The chain can be verified with a command-line tool. This is **tamper-evident, not tamper-proof**: it *detects* on-host modification rather than preventing it, so the store's file permissions should be locked down and the service run with least privilege. Because the hash chain lives on the same host as the data it protects, the audit trail (and the general log) can be **forwarded off-box** to a syslog or SIEM collector so an independent copy survives a host compromise; the same redaction filters apply to the forwarded stream, and the forwarding transport should be terminated at a local TLS agent or kept on a trusted management network.

Transport security

Native transport TLS ships for both the API/WebSocket plane and the MLLP data plane, with a reverse-proxy hardening path for TLS-terminating upstreams:

- **API / WebSocket TLS** — in-process TLS with a TLS 1.2+ floor and configurable certificates, cipher suites, and minimum version, plus an optional client CA for mTLS.
- **MLLP-over-TLS** — per connection, with a required server certificate, peer verification by default, and optional mutual TLS.
- **Outbound TLS verification fails closed** for LDAPS, database, and REST egress; it can be overridden only through an explicit insecure-TLS escape that logs a loud warning.

TLS is **off by default** because the default posture is loopback-only. As noted above, a non-loopback *plaintext* bind is refused at startup — so any deployment that exposes a listener beyond the local host must supply TLS, in-process or at a trusted upstream proxy.

Application and input security

All inbound HL7, file, and configuration content is treated as untrusted **data, never instructions**, and no message is ever silently dropped — failures are persisted as error or dead-letter dispositions.

- **Parsing** is two-tier: a tolerant parser on the hot path with opt-in strict validation, and size and segment caps enforced *before* parse (16 MiB / 10,000 segments).
- **Injection defenses are uniform** — parameterized SQL throughout, an ODBC connection-string guard, and RFC 4515 LDAP filter escaping.
- **Request and file limits** — an HTTP body cap with chunked-body rejection, a file-input cap with filename sanitization and path-traversal defense.
- **SSRF hardening** — REST destinations refuse redirects, and webhook alert sinks enforce an `http(s)` scheme with an optional host allowlist and a no-redirect opener.
- **DoS caps** — bounded MLLP frame size, connection count, and idle timeout; bounded file, HTTP, and WebSocket limits. The WebSocket Origin is checked against an allowlist before the connection is accepted, and interactive API docs are disabled by default.
- **Safe error responses** — clients receive generic `500` responses with no stack traces.

An opt-in, fail-closed **outbound egress allowlist** (for MLLP, HTTP, and file destinations), enforced at config load, reload, and start, constrains where PHI can be sent. It must be configured to take effect.

Secure development lifecycle and CI security gates

Engineering controls are a relative strength of the project, and they run as **blocking** gates in continuous integration — a violation fails the build:

- **Static analysis (SAST)** — Bandit plus custom Semgrep rules that forbid known-dangerous patterns (shell execution, `eval` / `exec`, insecure deserialization, disabled TLS verification).
- **Dependency auditing (SCA)** — `pip-audit` over a **hash-pinned lockfile** (so the audit is reproducible), a weekly scheduled run, a lockfile drift guard, and Dependabot updates.
- **Secret scanning** — full-history scanning, mirrored as a pre-commit check.

Behind the pipeline sit a documented per-interface threat model, a release gate, a root-cause-analysis template, and published remediation targets (Critical within 7 days, High within 30, Medium within 90). Security-critical design decisions are governed by Architecture Decision Records. Releases are tagged and **Sigstore-signed**, ship a per-release CycloneDX **SBOM**, and use PyPI Trusted Publishing (PEP 740 attestations) with GitHub-native SLSA build provenance — so the provenance of an installed artifact can be verified, not just its contents. A private vulnerability-disclosure channel with the published SLAs above is in place.

Honest caveats: development is currently single-maintainer, with CI gates and adversarial code review compensating for the absence of a second human reviewer; end-to-end hash-pinned *install* enforcement and GitHub Advanced Security code scanning are on the roadmap rather than in place today.

Threat model by interface

The project maintains a STRIDE-lite threat model with explicit trust boundaries. The trust boundary is the **local host**: the model assumes a trusted operating system with correct file permissions and operator-supplied volume encryption. The engine is **single-tenant** and makes no cross-tenant isolation guarantees. The table below summarizes the posture per interface.

Interface	Primary threats	Controls
Operator API / console	Credential theft, privilege escalation, session hijack	Required auth, deny-by-default RBAC, field-level PHI authorization, MFA + step-up on sensitive actions, opaque revocable sessions, loopback-by-default with TLS required off-loopback
Inbound HL7 / MLLP	Malformed-message DoS, parser abuse, untrusted-network exposure	Pre-parse size/segment caps, frame/connection/idle caps, MLLP-over-TLS with peer verification, no-silent-drop persistence
Inbound file	Path traversal, oversized input	Filename sanitization, path-traversal defense, input size cap
Inbound X12 / FHIR	Untrusted-payload injection, content confusion	Payload-agnostic ingress (formats are not force-applied), tolerant codecs, the same parameterization and persistence guarantees
Outbound REST / FHIR	SSRF, credential leakage, sending PHI to the wrong place	Redirect refusal, fail-closed egress allowlist, OAuth 2.0 client-credentials / SMART Backend Services, fail-closed TLS verification
Active Directory (LDAPS)	Credential interception, filter injection	LDAPS with certificate verification, RFC 4515 filter escaping, directory-side lockout and MFA
Database egress	Connection-string injection, MITM	Parameterized queries, ODBC injection guard, fail-closed TLS verification
Audit / log forwarding	On-host tampering, PHI leakage in transit	Hash-chained tamper-evidence, off-box forwarding for an independent copy, redaction applied to the forwarded stream

ASVS 5.0 Level 3 self-assessment

The project has completed a full **OWASP ASVS 5.0 Level 3 self-assessment** across the in-scope chapters, with a per-requirement verdict and an evidence path for each requirement.

Verdict	Count	Meaning
Met	212	Requirement self-assessed as met, with an evidence path.
Failed	0	None.
Partial	0	None.
Not applicable	133	Architecturally inapplicable (for example, no browser frontend and no WebRTC).
Total	345	The full Level 3 set across all chapters.

What "self-assessment" means here — read this part carefully. This is a point-in-time, code-backed gap analysis conducted by the project against the canonical ASVS text. It is **not** a certification, an accreditation, a passed audit, or a third-party penetration test, and it should not be read as one. At Level 3, ASVS *recommends* — but does not require — an independent code review and penetration test; those are **planned but have not yet been performed**. MessageFoundry **supports a HIPAA-compliant deployment** and maps its technical controls to the HIPAA Security Rule technical safeguards and to relevant NIST guidance, but a HIPAA-compliant deployment also depends on administrative and physical safeguards, and on disciplined operation (volume encryption, file permissions, key management, retention, and egress allowlisting) that are the deployer's responsibility. We describe what the engine does and what it does not do, and we publish our own residual risks rather than obscure them.

An independent security assessment is a documented prerequisite before any off-loopback or production PHI exposure.

Deployment responsibilities

The engine cannot enforce host-level controls, so the following remain the deployer's responsibility: full-volume encryption for the database, write-ahead files, temporary directories, and file-connector directories; host operating-system permissions and physical security; backup encryption and a secure backup lifecycle; and an operational incident-response and breach-notification program. Built-in backup/disaster-recovery tooling and an incident-response workflow are not part of the engine today; active-passive high availability is built and failover-validated on PostgreSQL and SQL Server.

Learn more

- **Secure Development Standards** — the full engineering detail behind the controls summarized here.
- **Security model, roles, and sessions** — docs/SECURITY.md
- **PHI handling and at-rest encryption** — docs/PHI.md
- **Vulnerability disclosure** — SECURITY.md
