Contents

1 Software Security - RSA Encryption Service SSDLC Analysis

1.1 Table of Contents

1.2 1. Introduction and Motivation

The development of a RESTful web service for RSA-based file encryption presents a complex security engineering challenge. This document provides a comprehensive analysis of the Secure Software Development Lifecycle (SSDLC) applied to this case study. We examine how security principles must be integrated at every phase - from requirements through maintenance - to build systems that resist emerging threats while maintaining practical usability.

This analysis serves multiple purposes:

1. **Educational Value**: Demonstrates real-world security engineering practices by examining a cryptographic service
2. **Risk Management**: Identifies threats and proposes mitigations for a production deployment scenario
3. **Standards Alignment**: Maps security practices to recognized frameworks including OWASP ASVS, NIST CSF, and ISO/IEC 27001
4. **Implementation Guidance**: Provides both secure and insecure code examples to illustrate common pitfalls

The RSA encryption service represents a security-critical system. Failures in cryptographic systems can result in data breaches, privacy violations, regulatory

non-compliance, and reputational damage. Therefore, security must be a primary design driver rather than an afterthought.

1.3 2. Security Requirements Engineering

1.3.1 2.1 Functional Requirements

F1: User Registration and Authentication - The system shall allow users to register with a username and password - The system shall generate unique RSA keypairs during registration - The system shall authenticate users using credentials and issue session tokens

F2: File Upload and Encryption - The system shall accept file uploads from authenticated users - The system shall encrypt uploaded files using the user's RSA public key - The system shall store encrypted files with metadata including filename and upload timestamp

F3: File Download and Decryption - The system shall allow authenticated users to download their encrypted files - The system shall decrypt files using the user's private key upon authorization - The system shall maintain audit records of all decryption operations

F4: File Management - The system shall allow users to list their uploaded files with associated metadata - The system shall support file deletion with audit logging - The system shall prevent users from accessing other users' files

F5: Key Management - The system shall generate and store RSA keypairs securely - The system shall support key rotation operations - The system shall never expose private keys to clients except during initial registration

1.3.2 2.2 Non-Functional Requirements

N1: Performance - File upload/download operations shall complete within 5 seconds for 100KB files - API response times shall remain below 200ms for 99th percentile under normal load - The system shall support at least 1,000 concurrent users

N2: Scalability - The system shall support horizontal scaling across multiple API servers - Database connections shall be pooled and reused efficiently - Encryption operations shall not become bottlenecks during peak usage

N3: Availability - The system shall maintain 99.9% uptime excluding maintenance windows - Database shall be replicated across multiple availability zones - Failover shall occur automatically within 30 seconds of primary failure

N4: Maintainability - Code shall follow established Python conventions (PEP 8) - All security-critical functions shall include comprehensive documentation - The system shall support graceful dependency updates

N5: Usability - API documentation shall provide clear examples for client integration - Error messages shall guide users toward corrective actions - Client libraries shall be available for common programming languages

1.3.3 2.3 Security Requirements

S1: Confidentiality - All file data shall be encrypted at rest using RSA encryption - Data in transit shall be protected using TLS 1.2 or higher - Private keys shall never be exposed to unauthorized parties

S2: Integrity - All encrypted files shall include cryptographic integrity protections - The system shall detect and reject tampered encrypted data - File modifications shall be logged and auditable

S3: Authentication - User authentication shall require both username and password - Multi-factor authentication shall be optional but available - Session tokens shall expire after 1 hour of inactivity

S4: Authorization - Access control decisions shall be enforced on all API endpoints - Users shall only access their own files and keys - Administrative operations shall require elevated privileges

S5: Audit and Accountability - All security-relevant events shall be logged including logins, file access, and key operations - Audit logs shall be stored securely and remain immutable - Logs shall be retained for minimum 90 days

S6: Non-repudiation - File encryption operations shall be attributable to specific users - Audit trails shall provide evidence of who performed which operations - Cryptographic signatures shall support dispute resolution

1.3.4 2.4 Conflicting Requirements and Prioritization

Several requirement conflicts emerge during security engineering:

Conflict 1: Security vs. Performance (Cryptographic Strength) - Requirement: S1 (use strong encryption) vs. N1 (rapid response times) - Analysis: RSA 2048-bit encryption operations require substantial computational resources. Larger key sizes (4096-bit) provide better long-term security but impact performance - Resolution: Use RSA 2048-bit as baseline with option for 4096-bit for highly sensitive deployments. Server-side implementation minimizes client-side overhead. Hybrid encryption (RSA + AES) improves performance while maintaining security - Priority: Security takes precedence; performance is secondary

Conflict 2: Usability vs. Security (Key Management) - Requirement: F5 (never expose private keys) vs. F1 (user convenience) - Analysis: Returning private keys to clients during registration improves usability but creates significant security risks (key exposure, insecure storage). Server-side key management is more secure but requires additional infrastructure - Resolution: Generate keys server-side and store securely. Clients authenticate using credentials only. Key

recovery uses multi-factor authentication - Priority: Security takes precedence; accept reduced usability

Conflict 3: Audit Logging vs. Privacy - Requirement: S5 (comprehensive logging) vs. privacy regulations (GDPR) - Analysis: Detailed logging of user actions may capture personally identifiable information or reveal sensitive usage patterns, creating privacy concerns - Resolution: Implement log minimization - record security events and access patterns but not file contents. Implement log retention policies. Allow users to request log deletion after compliance periods - Priority: Balance through careful log design; security slightly prioritized

Conflict 4: Scalability vs. Security (Stateless Design) - Requirement: N2 (horizontal scaling) vs. S3 (session management) - Analysis: Distributed systems require stateless design, limiting server-side session management capabilities - Resolution: Use JWT tokens with cryptographic verification. Implement token revocation lists for compromised tokens. No server state required for verification - Priority: Both satisfied through JWT architecture

Conflict 5: Availability vs. Security (Key Protection) - Requirement: N3 (99.9% uptime, replicated database) vs. S1 (key confidentiality) - Analysis: Database replication exposes keys to multiple systems. Database replication and backup create exposure windows - Resolution: Separate key storage from main database. Use hardware security modules (HSM) for production. Implement encryption key hierarchy - master keys in HSM, data keys in database - Priority: Security prioritized; implement proper key management

1.3.5 2.5 Standards and Framework Mapping

1.3.5.1 OWASP ASVS Level 3 Mapping

ASVS Control Area	Requirement	Status
Architecture	Secure design principles, defense-in-depth	S1, S4
Authentication	Strong credential verification, MFA support	S3, F1
Session Management	Token expiration, secure session handling	S3, F5
Validation	Input validation, sanitization	All endpoints
Encoding	Output encoding, prevention of injection	All responses
Cryptography	Proper algorithm selection, key management	S1, F5, S2

ASVS Control Area	Requirement	Status
Error Handling	Generic error messages, no information disclosure	All endpoints
Data Protection	Encryption at rest and transit, key protection	S1, S5
Communications	TLS enforcement, certificate validation	S1, N1

1.3.5.2 NIST Cybersecurity Framework (CSF) Mapping

NIST CSF Function	Implementation
IDENTIFY	Threat modeling (Section 3), asset inventory
PROTECT	Access controls (S4), encryption (S1), authentication (S3)
DETECT	Audit logging (S5), anomaly detection
RESPOND	Incident response procedures (Section 8)
RECOVER	Disaster recovery, key compromise procedures

1.3.5.3 ISO/IEC 27001 Controls

Control Area	Controls	Requirement
Organization	A.5 Leadership	SSDLC governance
People	A.6 HR Security	Developer training, security culture
Technology	A.10 Cryptography	S1, F5 encryption implementation
Technology	A.11 Physical	HSM placement, secure data centers
Operations	A.12 Operations	Backup, disaster recovery
Communications	A.13 Communications	TLS, secure channels
System Acquisition	A.14 Supplier	Dependency management
Incidents	A.16 Incidents	Incident response (Section 8)

1.4 3. Threat Modeling and Risk Assessment

1.4.1 3.1 STRIDE Threat Analysis

STRIDE methodology identifies six threat categories: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege.

1.4.1.1 Spoofing Threats **T1: User Identity Spoofing** - Attacker impersonates legitimate user by guessing/stealing credentials - Likelihood: High (credential attacks are common) - Impact: Critical (access to user's encrypted

files) - Mitigation: Enforce strong passwords, rate-limit login attempts, implement MFA - Residual Risk: Compromised credentials still enable access; MFA mitigates to Medium

T2: Service Spoofing - Attacker sets up fake encryption service to capture credentials and keys - Likelihood: Medium (requires network positioning) - Impact: Critical (all user data compromised) - Mitigation: HTTPS/TLS certificate validation, DNS security (DNSSEC) - Residual Risk: Certificate attacks remain (Medium)

T3: Cryptographic Key Spoofing - Attacker substitutes legitimate public key with malicious key - Likelihood: Medium (requires key distribution compromise) - Impact: Critical (files encrypted with attacker's key) - Mitigation: Public key infrastructure (PKI), certificate pinning - Residual Risk: Low after PKI implementation

1.4.1.2 Tampering Threats **T4: Encrypted File Tampering** - Attacker modifies encrypted file data in transit or storage - Likelihood: Medium (requires database or network access) - Impact: High (file corruption; possible decryption failure) - Mitigation: HMAC for integrity verification, digital signatures - Residual Risk: Low with Encrypt-then-MAC pattern

T5: Audit Log Tampering - Attacker modifies audit logs to hide malicious activity - Likelihood: Low (requires database access) - Impact: High (eliminates accountability, hinders incident response) - Mitigation: Write-once storage, log signature verification, off-site replication - Residual Risk: Very Low with immutable log storage

T6: Configuration Tampering - Attacker modifies system configuration to weaken security settings - Likelihood: Low (requires system access) - Impact: Critical (entire security posture compromised) - Mitigation: Configuration management, integrity verification, RBAC - Residual Risk: Low with proper access controls

T7: Private Key Tampering - Attacker modifies private keys to invalid state - Likelihood: Low (requires key storage access) - Impact: Critical (files become unrecoverable) - Mitigation: Key replication, HSM protection, regular integrity checks - Residual Risk: Low with HSM and backups

1.4.1.3 Repudiation Threats **T8: Encryption Operation Repudiation** - User denies uploading/encrypting specific files - Likelihood: Low (low motivational value) - Impact: Medium (accountability issues) - Mitigation: Digital signatures, audit logs with timestamps - Residual Risk: Low with proper audit trail

T9: Decryption Operation Repudiation - User denies accessing specific encrypted files - Likelihood: Low (low motivational value) - Impact: Medium

(legal/compliance issues) - Mitigation: Immutable audit logs, cryptographic proof of access - Residual Risk: Low with proper audit trail

1.4.1.4 Information Disclosure Threats **T10: Private Key Disclosure** - Private keys exposed to unauthorized parties through various vectors - Likelihood: Medium (multiple attack vectors) - Impact: Critical (complete compromise of user's files) - Vectors: Memory dumps, log files, client storage, network capture - Mitigation: Never transmit private keys, server-side key generation, secure deletion - Residual Risk: Low with server-side key management

T11: Encryption Key Material Leakage - Cryptographic materials (intermediate values, random number generator state) leak - Likelihood: Low (requires low-level access) - Impact: Critical (potential key recovery) - Mitigation: Use validated cryptographic libraries, prevent memory swapping, use secure enclaves - Residual Risk: Low with proper library selection and system hardening

T12: Metadata Information Disclosure - File names, timestamps, access patterns reveal sensitive information - Likelihood: Medium (metadata stored unencrypted) - Impact: Medium (privacy violation, pattern analysis) - Mitigation: Encrypt file metadata, minimize metadata collection - Residual Risk: Low with metadata encryption

T13: Error Message Information Disclosure - Error messages reveal system internals enabling further attacks - Likelihood: Medium (developers often overcommunicate errors) - Impact: Low-Medium (reconnaissance aid) - Mitigation: Generic error messages, detailed logs (not exposed to users) - Residual Risk: Low with proper error handling

T14: Traffic Analysis Information Disclosure - Attacker analyzes encrypted traffic patterns to infer usage - Likelihood: Medium (passive network observation) - Impact: Low-Medium (privacy concern) - Mitigation: Constant-rate padding, traffic shaping - Residual Risk: Medium (traffic analysis inherently difficult to prevent)

T15: Timing Attack Information Disclosure - Attacker uses decryption operation timing to infer key properties - Likelihood: Low (requires specialized knowledge) - Impact: Medium (potential key recovery) - Mitigation: Constant-time cryptographic implementations, padding oracle protection - Residual Risk: Low with OAEP and validated libraries

T16: Side-Channel Information Disclosure - Attacker exploits power consumption, electromagnetic emissions to recover keys - Likelihood: Very Low (requires specialized hardware) - Impact: Critical (key compromise) - Mitigation: Use validated cryptographic libraries, HSM for sensitive operations - Residual Risk: Very Low for typical cloud deployment

1.4.1.5 Denial of Service Threats **T17: Cryptographic Operation DoS** - Attacker floods service with encryption/decryption requests - Likelihood:

High (easy to execute) - Impact: Medium (service degradation) - Mitigation: Rate limiting, request throttling, computational cost awareness - Residual Risk: Low-Medium (acceptable degradation)

T18: Database Connection Exhaustion - Attacker opens many connections, exhausting connection pool - Likelihood: Medium (basic resource attack) - Impact: Medium (service unavailability) - Mitigation: Connection pooling, idle timeout, max connection limits - Residual Risk: Low with proper pooling

T19: Storage Space Exhaustion - Attacker uploads extremely large files consuming all storage - Likelihood: Medium (requires authentication only) - Impact: Medium (service degradation) - Mitigation: Per-user storage quotas, file size limits - Residual Risk: Low with quotas enforced

T20: Memory Exhaustion - Attacker triggers operations consuming excessive memory - Likelihood: Low (requires specific triggering conditions) - Impact: Medium (service crash) - Mitigation: Memory limits, input validation, efficient algorithms - Residual Risk: Low with proper resource management

T21: CPU Exhaustion via Cryptographic Operations - Attacker requests computationally expensive operations (large key generation) - Likelihood: Medium (RSA operations are CPU-intensive) - Impact: Medium (service slowdown) - Mitigation: Rate limiting, operation queuing, dedicated cryptographic resources - Residual Risk: Low-Medium (acceptable degradation with rate limiting)

1.4.1.6 Elevation of Privilege Threats

T22: Horizontal Privilege Escalation - Attacker accesses other users' files by manipulating file IDs - Likelihood: High (common API vulnerability) - Impact: Critical (breach of confidentiality) - Mitigation: Enforce access control on all endpoints, verify user ownership - Residual Risk: Low with proper authorization checks

T23: Vertical Privilege Escalation - Attacker escalates from regular user to administrator - Likelihood: Low (requires logic flaws) - Impact: Critical (complete system compromise) - Mitigation: RBAC implementation, role verification on sensitive operations - Residual Risk: Low with proper RBAC design

T24: JWT Token Privilege Escalation - Attacker modifies JWT token claims to elevate privileges - Likelihood: Low (requires key compromise or algorithm bypass) - Impact: Critical (privilege elevation) - Mitigation: Strong token verification, secure key management, algorithm enforcement - Residual Risk: Low with proper token handling

1.4.2 3.2 Attack Scenarios - In-Depth Analysis

1.4.2.1 Scenario 1: DBA Insider Threat with Privilege Abuse

Attack Vector: Database Administrator with legitimate access misuses privileges to access encrypted files.

Timeline: - DBA observes user credentials during troubleshooting - DBA creates backdoor account with administrative privileges - DBA logs in as backdoor user and downloads encrypted files - DBA exports private keys from database - DBA decrypts files offline and exfiltrates sensitive data

Likelihood: Medium (DBAs have legitimate access; motivation varies)

Impact: Critical (complete data breach, privacy violation)

Mitigation Strategy: - Implement least privilege - DBA should not have direct key access - Use separate key management service with dedicated administrators - Enforce multi-person approval for sensitive operations - Implement detailed audit logging of all DBA activities - Use hardware security modules (HSM) for key storage - Regular security audits of DBA activities - Segregate administrative roles (DBA, security admin, audit)

Residual Risk: Medium - Insider threats are inherently difficult to prevent; detection-focused approach recommended

1.4.2.2 Scenario 2: Supply Chain Attack - Compromised Dependency

Attack Vector: Attacker compromises cryptographic library used by service.

Timeline: - Attacker gains access to cryptographic library repository - Attacker introduces subtle vulnerability in encryption function (e.g., weak random number generation) - New library version published with vulnerability - Service upgrades to new version unaware of backdoor - Attacker can predict "random" components and break encryption

Likelihood: Low-Medium (supply chain attacks increasing in frequency)

Impact: Critical (complete cryptographic compromise)

Mitigation Strategy: - Use vetted cryptographic libraries (OpenSSL, libsodium, cryptography.io) - Implement dependency pinning with specific versions - Use software composition analysis (SCA) tools to detect vulnerabilities - Perform source code review of critical dependencies - Implement cryptographic integrity checks on library functions - Subscribe to security advisories for all dependencies - Maintain air-gapped test environments for dependency updates - Have rollback procedures for problematic updates

Residual Risk: Low-Medium - Inherent to open source; partially mitigated through process controls

1.4.2.3 Scenario 3: Padding Oracle Attack - Cryptographic Implementation Flaw

Attack Vector: Attacker exploits vulnerable encryption padding to recover plaintext without key.

Timeline: - Attacker identifies service uses PKCS#1 v1.5 padding (vulnerable) - Attacker intercepts encrypted file and attempts decryption - Attacker sends multiple decryption attempts with modified ciphertext - Service error responses

reveal padding validity (oracle) - Attacker uses statistical analysis to recover plaintext byte-by-byte - Attacker reconstructs file contents without private key

Likelihood: Medium (if PKCS#1 v1.5 used; low if OAEP used)

Impact: Critical (complete breach of file confidentiality)

Mitigation Strategy: - Use OAEP padding instead of PKCS#1 v1.5 - Implement hybrid encryption (RSA encrypts AES key; AES encrypts data) - Use authenticated encryption (Encrypt-then-MAC or AEAD modes) - Standardize error responses (don't distinguish padding errors) - Implement rate limiting on decryption attempts - Add timing attack mitigations (constant-time operations) - Security testing to detect oracle vulnerabilities

Residual Risk: Low - OAEP eliminates oracle vulnerability class

1.4.2.4 Scenario 4: Virtual Machine Escape and Key Extraction

Attack Vector: Attacker compromises cloud infrastructure and escapes virtual machine to access host memory.

Timeline: - Attacker gains foothold in application via zero-day vulnerability - Attacker exploits hypervisor vulnerability to escape VM - Attacker gains access to host physical memory - Attacker locates RSA private keys in memory - Attacker extracts key material and derives full private keys

Likelihood: Very Low (requires sophisticated attacker and zero-day)

Impact: Critical (complete cryptographic compromise)

Mitigation Strategy: - Use dedicated hosts (not shared VM infrastructure) - Implement hardware security modules (HSM) for key storage - Use secure enclaves (Intel SGX, ARM TrustZone) for cryptographic operations - Minimize key material in application memory - Implement secure key deletion (overwrite memory) - Use memory protection features (SMEP, SMAP) - Regular security hardening of host systems - Behavioral monitoring for suspicious memory access

Residual Risk: Very Low - Sophisticated attack with low likelihood; HSM mitigates effectively

1.4.2.5 Scenario 5: Brute Force Attack - Weak Password Implementation

Attack Vector: Attacker performs distributed brute force attack against user passwords.

Timeline: - Attacker obtains list of usernames (via user enumeration or data breach) - Attacker attempts to log in with common passwords - No rate limiting allows thousands of attempts per second - Weak password hashing (fast hashing algorithm) enables rapid testing - Attacker successfully guesses password for high-value user - Attacker logs in and accesses encrypted files

Likelihood: High (common attack; easy to execute)

Impact: High (account compromise leading to data breach)

Mitigation Strategy: - Enforce strong password policy (minimum 12 characters, complexity) - Implement rate limiting (3 failed attempts per minute) - Implement account lockout (temporary after 5 failures) - Use strong password hashing (bcrypt with work factor >10) - Implement multi-factor authentication (reduces password compromise impact) - Monitor for suspicious login patterns - Implement CAPTCHA for repeated failures - Security awareness training on password selection

Residual Risk: Low-Medium - Strong authentication mechanisms reduce risk; insider threats still possible

1.4.3 3.3 Risk Assessment Matrix

Risk Level determined by: Likelihood x Impact x Exploitability

Risk Level	Likelihood	Impact	Exploitability	Count
Critical	High	Critical	High	6-8
High	Medium	Critical	Medium	5-7
Medium	High	High	Medium	4-6
Low	Low	Medium	Low	3-5
Very Low	Very Low	Low	Very Low	1-2

Critical Risk Threats (require immediate mitigation): - T22: Horizontal Privilege Escalation - T10: Private Key Disclosure - T3: Cryptographic Key Spoofing - Scenario 2: Supply Chain Attack - Scenario 3: Padding Oracle Attack

High Risk Threats (require urgent mitigation): - T1: User Identity Spoofing - T2: Service Spoofing - Scenario 1: DBA Insider Threat - T17: Cryptographic Operation DoS

Medium Risk Threats (require planned mitigation): - T4, T5, T6: Tampering threats - T14: Traffic Analysis - T19: Storage Exhaustion - Scenario 5: Brute Force Attack

---

1.5 4. Secure Software Architecture and Design

1.5.1 4.1 System Architecture Overview

The RSA encryption service is designed using a multi-tier architecture with clear separation of concerns and security boundaries.

1.5.1.1 Architecture Layers **Layer 1: Client Layer** - Web browsers or client applications - Responsibility: Credential input, file selection, key storage

(client-side or HSM) - Security: HTTPS validation, certificate pinning (for mobile apps)

Layer 2: Network Security Layer - TLS/SSL termination - API Gateway with rate limiting and DDoS protection - Responsibility: Encrypted transport, request validation - Security: TLS 1.2+, strong cipher suites, certificate management

Layer 3: API Server Layer - Flask application instances - Stateless design enabling horizontal scaling - Responsibility: Authentication, authorization, business logic - Security: Token validation, access control, input validation

Layer 4: Cryptographic Operations Layer - Dedicated cryptographic functions - Hardware security module (HSM) for production key management - Responsibility: RSA encryption/decryption, key generation - Security: Secure random number generation, constant-time operations, key protection

Layer 5: Data Access Layer - SQLite database (development) or PostgreSQL (production) - Connection pooling and statement parameterization - Responsibility: Data persistence, query execution - Security: Parameterized queries (SQL injection prevention), encryption at rest

Layer 6: Audit and Monitoring Layer - Centralized logging (CloudWatch, ELK stack) - Security event monitoring and alerting - Responsibility: Audit trail, intrusion detection, compliance - Security: Immutable logs, secure transmission, access control

1.5.2 4.2 Trust Boundaries

A trust boundary exists between system components that are at different security levels.

Boundary 1: Client-to-Server - Untrusted: Client application (may be malicious) - Trusted: Server infrastructure - Controls: TLS encryption, authentication, input validation - Risk: Man-in-the-middle, client compromise

Boundary 2: Server-to-Database - Untrusted: Potential database compromise - Trusted: Application logic - Controls: Principle of least privilege, SQL injection prevention, encryption - Risk: DBA insider threat, SQL injection through parameter tampering

Boundary 3: Application-to-Cryptographic-Module - Untrusted: Main application memory - Trusted: Cryptographic library (validated, reviewed) - Controls: Secure API usage, key handling procedures - Risk: Side-channel attacks, improper key management

Boundary 4: Local-to-Cloud - Untrusted: Cloud provider infrastructure - Trusted: Encrypted data and access controls - Controls: Encryption at rest, customer-managed keys, VPC isolation - Risk: Provider access, physical security compromise

1.5.3 4.3 Attack Surface Analysis

An attack surface is the sum of all methods an attacker could exploit to compromise the system.

Attack Surface 1: API Endpoints (External) - Endpoints: /api/register, /api/login, /api/upload, /api/decrypt, /api/files - Risk: Injection attacks, authorization bypass, resource exhaustion - Mitigation: Input validation, comprehensive authorization checks, rate limiting - Exposure: High (public-facing)

Attack Surface 2: Authentication Mechanism (External) - Mechanism: Username/password, JWT tokens, MFA - Risk: Credential stuffing, token compromise, MFA bypass - Mitigation: Strong password enforcement, secure token storage, MFA hardening - Exposure: High (public-facing)

Attack Surface 3: File Upload Handler (External) - Handler: /api/upload endpoint - Risk: Oversized files, malicious payloads, directory traversal - Mitigation: File size limits, type validation, secure file handling - Exposure: High (authenticated)

Attack Surface 4: Cryptographic Operations (Internal) - Operations: RSA encryption/decryption, key generation - Risk: Weak randomness, side-channel attacks, implementation flaws - Mitigation: Validated libraries, constant-time operations, security testing - Exposure: Medium (internal, cryptographic-specific)

Attack Surface 5: Database Interface (Internal) - Interface: SQL queries, connection management - Risk: SQL injection, unauthorized access, information disclosure - Mitigation: Parameterized queries, least privilege, encryption - Exposure: Medium (internal, admin-accessible)

Attack Surface 6: Configuration Management (Administrative) - Configuration: API keys, encryption parameters, feature flags - Risk: Configuration tampering, secrets exposure, weak defaults - Mitigation: Secure configuration storage, access control, audit logging - Exposure: Low (administrative access required)

Attack Surface 7: Deployment Infrastructure (Physical/Cloud) - Infrastructure: Servers, databases, load balancers, key management - Risk: Physical access, cloud provider compromise, misconfiguration - Mitigation: Cloud provider security controls, network segmentation, HSM - Exposure: Low (provider-managed)

1.5.4 4.4 Security Principles Application

1.5.4.1 Principle 1: Defense-in-Depth Defense-in-depth deploys multiple security layers so that if one fails, others remain.

Implementation:

Layer 1 - Network: API Gateway with DDoS protection, rate limiting **Layer 2 - Transport**: TLS 1.2+ encryption, certificate validation **Layer 3 - Authentication**: Strong credential requirements, MFA support **Layer 4 - Authorization**: Role-based access control, per-resource checks **Layer 5 - Cryptography**: RSA 2048-bit, OAEP padding, Encrypt-then-MAC **Layer 6 - Data Protection**: Encryption at rest, secure deletion **Layer 7 - Audit**: Comprehensive logging, anomaly detection

Failure Tolerance Example: If encryption key is compromised, access controls and audit logs still provide some protection and enable incident response.

1.5.4.2 Principle 2: Least Privilege

Least privilege restricts permissions to minimum necessary for function.

Implementation:

Database User Permissions:

```
- Application account: SELECT, INSERT, UPDATE on app tables only
- DBA account: Full access to system tables only (limited key access)
- Audit account: SELECT-only on audit logs
```

API Authorization:

```
- Regular users: Access own files and keys only
- Administrators: Limited admin functions (user management, not direct data access)
- Service accounts: Specific operations only (no user impersonation)
```

File System Permissions:

```
- Application process: Read application code, write logs, read configs
- Database process: Read/write database files, no access to application code
- Administrators: Full access with audit logging
```

1.5.4.3 Principle 3: Fail-Secure (Secure-by-Default)

Fail-secure ensures that when systems fail, they fail in a secure state.

Implementation:

Access Control Failures: - Default: Deny all access - Procedure: Explicitly grant access only when properly authorized - Error: Request rejected rather than exposed

Cryptographic Failures: - Default: Operation fails and rejects request - Procedure: No data exposed, attempt logged - Recovery: Admin intervention required

Dependency Failures: - Default: Service unavailable rather than degraded - Procedure: Health checks verify critical dependencies - Recovery: Graceful shutdown to prevent data corruption

1.5.4.4 Principle 4: Separation of Concerns

Separation of concerns divides system into independent components.

Components:

Authentication Module: Handles credential verification and token generation **Authorization Module**: Enforces access control decisions **Cryptography Module**: Performs encryption/decryption operations **Persistence Module**: Manages database operations **Audit Module**: Records security-relevant events **API Module**: Handles HTTP request/response

Benefit: Each component can be tested, updated, and secured independently without affecting others.

1.5.5 4.5 Scalability and Maintainability Trade-offs

1.5.5.1 Scalability Considerations

Horizontal Scaling Requirements: - Stateless API design: No server-side session state (use JWT tokens) - Database replication: Read replicas for query scaling - Cache layer: Redis for frequently accessed data (public keys) - Asynchronous operations: Background job queue for long-running tasks

Potential Bottlenecks: - Cryptographic operations: RSA operations are CPU-intensive - Database writes: Audit logging creates write contention - Key management: HSM access limited to certain throughput

Scaling Solutions: - Dedicated cryptographic servers with GPU acceleration - Write-ahead logging for audit data with async replication - HSM with high-availability clustering - API Gateway with intelligent request routing

1.5.5.2 Maintainability Considerations

Code Organization: - Modular design with clear responsibility boundaries - Comprehensive documentation of security decisions - Automated testing of security-critical functions - Code review process focusing on security implications

Dependency Management: - Minimal dependencies to reduce attack surface - Regular security updates and testing - Software composition analysis (SCA) scanning - Explicit version pinning for reproducibility

Operational Procedures: - Clear deployment processes with security checks - Runbooks for common operational tasks - Incident response procedures with security focus - Regular security training for operations team

---

1.6 5. Authentication and Authorization

1.6.1 5.1 Multi-Tier Authentication Model

The service implements three-tier authentication providing defense-in-depth:

1.6.1.1 Tier 1: Credential-Based Authentication (Login) **Purpose**: Initial user verification

Mechanism: - Username and password submission via HTTPS - Password hashed using bcrypt with configurable work factor - Salted hashing prevents rainbow table attacks - Account lockout after 5 failed attempts (15-minute lockout)

Flow:

```
1. User submits credentials to /api/login
2. Server retrieves password hash from database
3. Server compares submitted password to stored hash
4. Match: Proceed to Tier 2
5. No match: Log attempt, increment failure counter, return generic error
```

Security Properties: - Resistant to: Brute force (rate limiting), dictionary attacks (bcrypt cost) - Vulnerable to: Credential stuffing (requires compromise elsewhere)

1.6.1.2 Tier 2: Token-Based Authentication (Session) **Purpose**: Ongoing authentication without re-entering credentials

Mechanism: - JSON Web Tokens (JWT) issued upon successful login - Token contains user ID, issued-at timestamp, expiration timestamp - Token signed with HMAC-SHA256 to prevent tampering - Expiration: 1 hour from issuance - No server-side session storage (stateless)

Flow:

```
1. Successful login returns JWT token
2. Client includes token in Authorization header: "Bearer <token>"
3. Server verifies token signature using SECRET_KEY
4. Server extracts user ID from token claims
5. Token accepted if signature valid and not expired
```

Token Structure:

```
Header: {"alg": "HS256", "typ": "JWT"}
Payload: {"user_id": 123, "iat": 1234567890, "exp": 1234571490}
Signature: HMAC-SHA256(header.payload, SECRET_KEY)
```

Security Properties: - Resistant to: Session hijacking (cryptographic verification), token forgery (signature) - Vulnerable to: Key compromise (forces token re-issuance for all users)

1.6.1.3 Tier 3: Multi-Factor Authentication (MFA) - Optional Enhancement **Purpose**: Enhanced security for high-risk operations

Mechanism: - Time-based One-Time Password (TOTP) using HMAC-TOTP - User configures authenticator app during registration - 6-digit code required for

sensitive operations (decrypt, delete) - Time window: 30-second validity (current and previous period)

Implementation Example:

```
@app.route('/api/decrypt/<int:file_id>', methods=['POST'])
@token_required
def decrypt_with_mfa(file_id):
    data = request.get_json()
    mfa_code = data.get('mfa_code')

    # Verify MFA code
    if not verify_totp(request.user_id, mfa_code):
        log_audit(request.user_id, 'MFA_FAILED')
        return {'error': 'Invalid MFA code'}, 401

    # Proceed with decryption
    ...
```

Security Properties: - Resistant to: Credential compromise (requires physical device) - Vulnerable to: SIM swapping (requires MFA recovery codes)

1.6.2 5.2 Authorization Framework - Role-Based Access Control (RBAC)

1.6.2.1 Role Definitions

User Role: - Permissions: Upload files, download own files, view own audit logs - Operations: /api/upload, /api/files, /api/decrypt (own files) - Restrictions: Cannot access other users' data, limited admin operations

Administrator Role: - Permissions: All user permissions plus user management, system configuration - Operations: All user operations plus /api/admin/* endpoints - Restrictions: Still subject to audit logging, cannot bypass encryption

1.6.2.2 Authorization Checks

Per-Resource Authorization:

Every resource access must verify both: 1. User is authenticated (valid JWT token) 2. User has permission for specific resource

Example - File Download:

```
@app.route('/api/decrypt/<int:file_id>', methods=['POST'])
@token_required
def decrypt_file(file_id):
    # Step 1: Verify authentication (decorator)
    # user_id is set by @token_required decorator

    # Step 2: Verify authorization (resource ownership)
    conn = sqlite3.connect(DATABASE)
```

```
    cursor = conn.cursor()
    cursor.execute(
        'SELECT user_id FROM files WHERE id = ? AND user_id = ?',
        (file_id, request.user_id)
    )
    file = cursor.fetchone()
    conn.close()

    if not file:
        log_audit(request.user_id, 'UNAUTHORIZED_ACCESS_ATTEMPT', f'File: {file_id}')
        return {'error': 'File not found'}, 404

    # Step 3: Perform operation
    ...
```

Authorization Matrix:

Resource	User	Admin	Anonymous
Own files	Read, Write, Delete	Same + others'	None
Other users' files	Deny	Read (audit)	None
User management	View self	Manage all	None
System configuration	None	Modify	None
Audit logs	Own logs	All logs	None

1.6.3 5.3 Privilege Escalation Analysis

1.6.3.1 Horizontal Privilege Escalation

Scenario: Regular user accesses other users' files

Attack Vector: Manipulate file_id parameter

```
User A:
POST /api/decrypt/1 (own file) - ALLOWED
POST /api/decrypt/2 (User B's file) - ATTACKED

Expected: 404 or 403
Vulnerable: If no ownership check, file decrypts with User A's key
```

Mitigation:

Current Implementation (VULNERABLE):

```
cursor.execute('SELECT encrypted_data FROM files WHERE id = ?', (file_id,))
```

Fixed Implementation:

```
cursor.execute(
    'SELECT encrypted_data FROM files WHERE id = ? AND user_id = ?',
```

```
    (file_id, request.user_id)
)
```

Impact: Critical - Users could access any file in system

1.6.3.2 Vertical Privilege Escalation

Scenario: Regular user gains administrator privileges

Attack Vector: JWT token modification or database manipulation

```
Legitimate token: {"user_id": 123, "role": "user"}
Attacker creates: {"user_id": 123, "role": "admin"}
Attacker signs with own key
```

Mitigation:

1. Token verification ensures signature matches SECRET_KEY
2. Only server can issue valid tokens (user cannot self-issue)
3. Audit logging detects admin token usage by regular user

```python
# Token issued by server during login
payload = {'user_id': user_id, 'role': user_role}
token = jwt.encode(payload, SECRET_KEY, algorithm='HS256')

# Token verified by server
decoded = jwt.decode(token, SECRET_KEY, algorithms=['HS256'])
# Signature doesn't match if attacker used different key
```

Impact: Critical - Attacker gains full system access

1.6.3.3 Capability Escalation

Scenario: User escalates specific capability beyond intended scope

Attack Vector: Exploit missing authorization checks

```
User attempts: POST /api/admin/create-user (admin-only)
Expected: 403 Forbidden
Vulnerable: If no @admin_required decorator, user could create accounts
```

Mitigation: Explicit capability checks on all sensitive operations

```python
def admin_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        if not user_has_role(request.user_id, 'admin'):
            log_audit(request.user_id, 'UNAUTHORIZED_CAPABILITY_ATTEMPT')
            return {'error': 'Forbidden'}, 403
        return f(*args, **kwargs)
    return decorated

@app.route('/api/admin/users', methods=['POST'])
```

```
@token_required
@admin_required
def create_user():
    # Only admin can reach here
    ...
```

1.6.4 5.4 Access Control Testing Strategy

1.6.4.1 Unit Tests for Authorization

```
def test_user_cannot_access_other_user_files():
    # Create User A and User B
    user_a_token = login('user_a', 'password_a')
    user_b_token = login('user_b', 'password_b')

    # User A uploads file
    file_id = upload_file(user_a_token, 'secret.txt')

    # User B attempts to decrypt User A's file
    response = decrypt_file(user_b_token, file_id)

    # Assertion: Access denied
    assert response.status_code == 404

    # Assertion: Attempt logged
    assert audit_log_contains('user_b', 'UNAUTHORIZED_ACCESS_ATTEMPT')

def test_admin_cannot_bypass_encryption():
    admin_token = login('admin', 'admin_password')
    user_file_id = 123  # User's encrypted file

    # Admin requests file
    response = get_file(admin_token, user_file_id)

    # Assertion: File returned encrypted, not decrypted
    assert response.data is encrypted
    # Admin needs user's private key to decrypt, which admin shouldn't have
```

1.6.4.2 Integration Tests for Authorization Flows

```
def test_complete_authentication_authorization_flow():
    # 1. Register new user
    register('newuser', 'strongpassword123')

    # 2. Login - obtain token
    token = login('newuser', 'strongpassword123')
```

```
    # 3. Access protected resource - should succeed
    files = list_files(token)
    assert files != None

    # 4. Access with expired token - should fail
    old_token = generate_expired_token()
    files = list_files(old_token)
    assert files == 401

    # 5. Access with invalid token - should fail
    files = list_files('invalid.token.here')
    assert files == 401
```

1.7 6. Secure Implementation and Code Assurance

1.7.1 6.1 Cryptographic Implementation Flaws

1.7.1.1 Flaw 1: PKCS#1 v1.5 Padding - Vulnerable to Padding Oracle Attacks Description:

PKCS#1 v1.5 padding adds random bytes to pad message to RSA key size. The decryption process verifies padding format, revealing through error messages whether padding is valid - this is an "oracle" enabling attackers to recover plaintext.

Vulnerable Code:

```
def decrypt_file(encrypted_data_b64, private_key_pem):
    private_key = serialization.load_pem_private_key(...)
    encrypted_data = base64.b64decode(encrypted_data_b64)

    # VULNERABLE: PKCS#1 v1.5 padding
    plaintext = private_key.decrypt(
        encrypted_data,
        padding.PKCS1v15()
    )

    return plaintext
```

Attack Explanation:

1. Attacker obtains encrypted message C = RSA_encrypt(M)
2. Attacker sends decryption requests: C, 2C, 3C, ... (modified ciphertexts)
3. For each request, server returns: "Valid padding" or "Invalid padding"
4. Attacker uses statistical analysis to determine original plaintext
5. Attacker recovers full message without private key

Secure Implementation:

```
def decrypt_file_secure(encrypted_data_b64, private_key_pem):
    private_key = serialization.load_pem_private_key(...)
    encrypted_data = base64.b64decode(encrypted_data_b64)

    # SECURE: OAEP padding prevents oracle attacks
    plaintext = private_key.decrypt(
        encrypted_data,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    return plaintext
```

Why OAEP is Secure: - Deterministic randomization (random padding cannot reveal plaintext) - Padding verification doesn't leak information (fail fast) - Mathematically proven security against oracle attacks

Impact of Vulnerability: Critical - Complete plaintext recovery possible

1.7.1.2 Flaw 2: Direct RSA Encryption on Large Files Description:

RSA can only encrypt data smaller than key size (256 bytes for 2048-bit key). Using RSA directly on files fails for most real-world scenarios.

Vulnerable Code:

```
def encrypt_file(file_data, public_key_pem):
    public_key = serialization.load_pem_public_key(...)

    # VULNERABLE: Direct RSA encryption truncates file to 190 bytes
    ciphertext = public_key.encrypt(
        file_data[:190],  # Truncation!
        padding.OAEP(...)
    )

    return base64.b64encode(ciphertext)
```

Problems: - Only first 190 bytes encrypted (rest lost) - No integrity protection (ciphertext unverified) - No confidentiality for file size (observable in ciphertext) - Inefficient (RSA operations expensive for each block)

Secure Implementation - Hybrid Encryption:

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

def encrypt_file_secure(file_data, public_key_pem):
```

```
public_key = serialization.load_pem_public_key(...)

# Step 1: Generate random AES key
aes_key = os.urandom(32)  # 256-bit key

# Step 2: Encrypt AES key with RSA
encrypted_aes_key = public_key.encrypt(
    aes_key,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)

# Step 3: Encrypt file data with AES
iv = os.urandom(16)
cipher = Cipher(
    algorithms.AES(aes_key),
    modes.CBC(iv),
    backend=default_backend()
)
encryptor = cipher.encryptor()

# Apply PKCS7 padding for AES
padded_data = file_data + b'\x00' * (16 - len(file_data) % 16)
ciphertext = encryptor.update(padded_data) + encryptor.finalize()

# Step 4: Compute HMAC for integrity
h = hmac.HMAC(aes_key, hashes.SHA256(), backend=default_backend())
h.update(iv + ciphertext)
tag = h.finalize()

# Return: encrypted_key + iv + ciphertext + tag
return base64.b64encode(
    encrypted_aes_key + iv + ciphertext + tag
)
```

Why Hybrid Encryption is Secure: - RSA encrypts only key (small, suitable for RSA) - AES encrypts data (efficient, suitable for large files) - HMAC provides integrity verification - No truncation or data loss

Impact of Vulnerability: Critical - Data loss and confidentiality breach

1.7.1.3 Flaw 3: Missing Integrity Protection Description:

Encryption without integrity verification allows attackers to modify ciphertexts

undetected, potentially causing decryption failures or logic errors.

Vulnerable Code:

```
# Encrypt file, transmit over network
encrypted = public_key.encrypt(aes_key, padding.OAEP(...))

# Attacker modifies ciphertext in transit
# Receiver decrypts and receives garbage or error
decrypted = private_key.decrypt(modified_ciphertext, padding.OAEP(...))
# No verification that ciphertext wasn't modified
```

Secure Implementation - Encrypt-then-MAC:

```
import hmac
from cryptography.hazmat.primitives import hashes

def encrypt_with_integrity(data, key):
    # Step 1: Encrypt
    iv = os.urandom(16)
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    encryptor = cipher.encryptor()
    ciphertext = encryptor.update(data) + encryptor.finalize()

    # Step 2: Compute MAC over ciphertext
    h = hmac.HMAC(key, hashes.SHA256())
    h.update(iv + ciphertext)
    tag = h.finalize()

    return iv + ciphertext + tag

def decrypt_with_integrity_check(data, key):
    iv = data[:16]
    ciphertext = data[16:-32]
    tag = data[-32:]

    # Step 1: Verify MAC
    h = hmac.HMAC(key, hashes.SHA256())
    h.update(iv + ciphertext)
    h.verify(tag)  # Raises InvalidSignature if modified

    # Step 2: Decrypt
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    decryptor = cipher.decryptor()
    plaintext = decryptor.update(ciphertext) + decryptor.finalize()

    return plaintext
```

Why Encrypt-then-MAC is Secure: - MAC computed over ciphertext (changes detected immediately) - Decryption rejected if MAC invalid (prevents garbage decryption) - Prevents tampering at any point in the system

Impact of Vulnerability: High - Data integrity cannot be verified

1.7.2 6.2 Secure Coding Practices

1.7.2.1 Practice 1: Secure Random Number Generation Insecure:

```
import random
key = ''.join(random.choice(string.ascii_letters) for _ in range(32))
```

Problems: `random` module uses predictable PRNG, not cryptographically secure.

Secure:

```
import os
key = os.urandom(32)  # Uses system's CSPRNG (e.g., /dev/urandom)
```

1.7.2.2 Practice 2: Secure Password Hashing Insecure:

```
import hashlib
password_hash = hashlib.sha256(password.encode()).hexdigest()
```

Problems: Fast hashing allows brute force; no salt in this example.

Secure:

```
from werkzeug.security import generate_password_hash, check_password_hash
password_hash = generate_password_hash(password, method='pbkdf2:sha256')
check_password_hash(password_hash, password)
```

Benefits: PBKDF2 with salt, configurable iterations, resistant to GPU attacks.

1.7.2.3 Practice 3: Input Validation Insecure:

```
@app.route('/search')
def search():
    query = request.args.get('q')
    results = db.execute(f"SELECT * FROM files WHERE name LIKE '%{query}%'")
```

Problems: SQL injection vulnerability (concatenation).

Secure:

```
@app.route('/search')
def search():
    query = request.args.get('q')
    # Parameterized query prevents injection
    results = db.execute(
        "SELECT * FROM files WHERE name LIKE ?",
```

```
        (f'%{query}%',)
    )
```

1.7.2.4 Practice 4: Secure Error Handling Insecure:

```
try:
    decrypt_file(encrypted_data, private_key)
except Exception as e:
    return {'error': str(e), 'stack_trace': traceback.format_exc()}
```

Problems: Exposes internal details, enables reconnaissance.

Secure:

```
try:
    decrypt_file(encrypted_data, private_key)
except cryptography.exceptions.InvalidSignature:
    logger.error(f"Decryption failed for user {user_id}")
    return {'error': 'Decryption failed'}, 500
```

1.7.2.5 Practice 5: Secure Dependency Management Process:

```
1. Identify and document all dependencies
2. Pin to specific versions (not floating/ranges)
3. Regularly scan for vulnerabilities (Bandit, Safety)
4. Update only after security testing
5. Monitor security advisories continuously
```

requirements.txt:

```
Flask==2.3.0           # Specific version, not >=2.0.0
cryptography==41.0.0   # Specific version for reproducibility
PyJWT==2.8.0
Werkzeug==2.3.0
```

Scanning with Safety:

```
pip install safety
safety check --file requirements.txt
```

1.7.3 6.3 Code Review Checklist

Security code reviews should verify:

1. **Cryptography**
   - ☐ Using industry-standard algorithms (RSA 2048+, AES-256)
   - ☐ Using standard implementations (not custom crypto)
   - ☐ Proper padding schemes (OAEP not PKCS#1 v1.5)
   - ☐ Integrity protection (Encrypt-then-MAC or AEAD)
   - ☐ Secure random number generation (os.urandom, not random)
2. **Authentication**

- ☐ Password hashing with salt and cost factor
- ☐ Token expiration enforced
- ☐ No credentials in logs or error messages
- ☐ Rate limiting on login attempts

3. **Authorization**
   - ☐ All sensitive operations protected with auth check
   - ☐ Ownership verified before resource access
   - ☐ Role/permission verified for administrative operations
   - ☐ Horizontal privilege escalation prevented
4. **Data Protection**
   - ☐ Sensitive data encrypted in transit (HTTPS)
   - ☐ Sensitive data encrypted at rest
   - ☐ Secure deletion of sensitive data after use
   - ☐ No PII in audit logs
5. **Input Validation**
   - ☐ All user input validated for type and length
   - ☐ SQL injection prevented (parameterized queries)
   - ☐ Command injection prevented (no shell execution)
   - ☐ Path traversal prevented
6. **Error Handling**
   - ☐ Generic error messages to clients
   - ☐ Detailed errors logged server-side
   - ☐ No stack traces exposed to users
   - ☐ No information disclosure in error details
7. **Logging and Monitoring**
   - ☐ All security events logged (login, file access, errors)
   - ☐ Logs secure and access controlled
   - ☐ Sensitive data not logged
   - ☐ Monitoring configured for anomalies
8. **Dependency Management**
   - ☐ Dependencies documented and justified
   - ☐ Versions pinned for reproducibility
   - ☐ Vulnerability scanning enabled
   - ☐ Update procedures established

1.8 7. Security Testing, Validation, and Compliance

1.8.1 7.1 SSDLC Testing Phases

1.8.1.1 Phase 1: Requirements Validation

Objective: Verify security requirements are testable

Test Cases:

- Verify authentication requirement: "System shall authenticate users"
  - Test: Valid credentials accepted; invalid rejected

  - Test: Failed login attempts rate-limited
  - Test: Session tokens expire after 1 hour
- Verify authorization requirement: "Users shall access only own files"
  - Test: User cannot list other users' files
  - Test: User cannot decrypt other users' files
  - Test: Audit log shows attempted unauthorized access

1.8.1.2 Phase 2: Design Validation

Objective: Verify design implements requirements securely

Test Cases:

- Architecture: Verify trust boundaries
  - Test: TLS protects client-to-server communication
  - Test: Database encryption protects data at rest
  - Test: API Gateway enforces rate limiting
- Cryptography: Verify algorithm selection
  - Test: RSA 2048-bit used (not weaker)
  - Test: OAEP padding used (not PKCS#1 v1.5)
  - Test: HMAC-SHA256 used for integrity

1.8.1.3 Phase 3: Implementation Testing

Objective: Verify code implements design securely

Unit Tests:

```
def test_rsa_encryption_decryption():
    private_key = generate_rsa_keypair()
    public_key_pem = serialize_public_key(private_key)
    private_key_pem = serialize_private_key(private_key)

    plaintext = b"Sensitive data"
    encrypted = encrypt_file(plaintext, public_key_pem)
    decrypted = decrypt_file(encrypted, private_key_pem)

    assert decrypted[:len(plaintext)] == plaintext

def test_password_hashing_verification():
    password = "TestPassword123"
    hash = generate_password_hash(password)

    # Correct password accepted
    assert check_password_hash(hash, password)

    # Wrong password rejected
    assert not check_password_hash(hash, "WrongPassword")
```

```python
def test_jwt_token_expiration():
    user_id = 1
    token = generate_jwt_token(user_id)

    # Valid token verifies
    assert verify_jwt_token(token) == user_id

    # Expired token rejects (simulate expiration)
    old_token = jwt.encode(
        {'user_id': user_id, 'exp': datetime.datetime.utcnow() - datetime.timedelta(hours=1)
        SECRET_KEY
    )
    assert verify_jwt_token(old_token) is None
```

Security-Specific Tests:

```python
def test_sql_injection_prevention():
    # Attempt SQL injection in file search
    malicious_query = "'; DROP TABLE files; --"
    response = search_files(malicious_query)

    # Table should still exist (injection prevented)
    conn = sqlite3.connect(DATABASE)
    cursor = conn.cursor()
    cursor.execute("SELECT name FROM sqlite_master WHERE type='table' AND name='files'")
    assert cursor.fetchone() is not None

def test_authorization_horizontal_escalation_prevention():
    user_a_token = login('user_a', 'pass_a')
    user_b_file_id = 999  # File belonging to User B

    # User A attempts to decrypt User B's file
    response = decrypt(user_a_token, user_b_file_id)

    # Access denied
    assert response.status_code == 404
```

1.8.1.4 Phase 4: Deployment Validation

Objective: Verify security in production environment

Test Cases:

- TLS Configuration
  - Test: HTTPS enforced (redirect HTTP to HTTPS)
  - Test: TLS 1.2+ only (no SSL 3.0 or TLS 1.0)
  - Test: Strong cipher suites configured
  - Test: Certificate valid and properly installed

- Access Controls
  - Test: Database accessible only from application servers
  - Test: SSH access restricted to authorized IPs
  - Test: AWS IAM policies follow least privilege
- Monitoring
  - Test: Security events logged and alerting enabled
  - Test: Audit logs stored and immutable
  - Test: Anomaly detection rules configured

1.8.2 7.2 Static Application Security Testing (SAST)

Static analysis tools examine source code without execution to identify vulnerabilities.

1.8.2.1 Tool 1: Bandit (Python-specific)

Detects: Hardcoded passwords, dangerous functions, weak cryptography

```
bandit -r .
```

Output:

```
WARNING: Hardcoded SQL password
  File: database.py, Line 42
  Issue: db.connect('host', 'user', 'password123')

WARNING: Using insecure random
  File: crypto.py, Line 18
  Issue: random.choice() used instead of os.urandom()

HIGH: Weak hash function
  File: auth.py, Line 25
  Issue: hashlib.md5() - use SHA256 instead
```

1.8.2.2 Tool 2: Semgrep (Multi-language)

Detects: Custom rule-based patterns, vulnerable patterns

```
semgrep --config=p/security-audit .
```

1.8.3 7.3 OWASP Top 10 Vulnerability Mapping

OWASP Vulnerability	Risk	Mitigation
A1: Broken Access Control	Critical	Authorization checks, audit logging
A2: Cryptographic Failures	Critical	OAEP, Encrypt-then-MAC, secure key management

OWASP Vulnerability	Risk	Mitigation
A3: Injection	High	Parameterized queries, input validation
A4: Insecure Design	High	Security requirements, threat modeling, design review
A5: Security Misconfiguration	Medium	Secure defaults, configuration audit
A6: Vulnerable Components	High	Dependency scanning, updates
A7: Authentication Failures	Critical	MFA, rate limiting, strong passwords
A8: Software/Data Integrity	Medium	Code signing, package verification
A9: Logging Deficiencies	Medium	Comprehensive logging, audit trails
A10: SSRF	Low	URL validation, network segmentation

1.8.4 7.4 Compliance Requirements

1.8.4.1 GDPR Compliance Relevant Requirements: - Data minimization: Only collect necessary data - Purpose limitation: Use data only for stated purpose - Storage limitation: Delete data after retention period

Implementation: - Minimize user metadata collection - Encrypt PII - Implement data retention and deletion policies - Document processing activities (Data Processing Agreement)

1.8.4.2 HIPAA Compliance (if handling health data) Relevant Requirements: - Encryption of electronic protected health information (ePHI) - Access controls and audit logging - Incident response procedures

Implementation: - Encrypt all data at rest and in transit - Implement RBAC - Comprehensive audit logging - Regular security assessments

1.8.4.3 PCI-DSS Compliance (if handling payment data) Relevant Requirements: - Install and maintain firewall - Protect cardholder data - Implement strong authentication - Regular vulnerability assessments

Implementation (if applicable): - Network segmentation - Strong encryption - MFA for administrative access - Quarterly penetration testing

---

1.9 8. Deployment, Operations, and Incident Response

1.9.1 8.1 Cloud Deployment Architecture (AWS)

1.9.1.1 VPC Architecture Network Design:

```
AWS Account
  VPC (10.0.0.0/16)
    Public Subnets (10.0.1.0/24, 10.0.2.0/24)
      - NAT Gateway
      - API Gateway

    Private Subnets (10.0.10.0/24, 10.0.11.0/24)
      - Application Servers (Auto Scaling Group)
      - Network: ALB -> Flask apps

    Isolated Subnets (10.0.20.0/24, 10.0.21.0/24)
      - RDS Database (Multi-AZ)
      - ElastiCache Redis
      - AWS KMS endpoints

    VPC Endpoints
      - S3 gateway endpoint (for backups)
      - KMS interface endpoint
```

Security Groups:

```
ALB Security Group:
  Inbound: Port 443 (HTTPS) from 0.0.0.0/0
  Outbound: Port 5000 to Application SG

Application SG:
  Inbound: Port 5000 from ALB SG
  Outbound: Port 3306 to Database SG, 443 to Internet (updates)

Database SG:
  Inbound: Port 3306 from Application SG
  Outbound: None (no outbound required)
```

1.9.2 8.2 Secret Management

Secrets Stored in AWS Secrets Manager:

```
{
  "db_password": "randomly_generated_password_xyz",
  "jwt_secret_key": "cryptographically_random_key",
  "rsa_master_key": "encrypted_master_key_id",
  "api_keys": {
    "external_service_1": "api_key_value",
```

```
    "external_service_2": "api_key_value"
  }
}
```

Rotation Policy: - Database passwords: Rotate every 90 days - API keys: Rotate every 30 days - JWT secret: Rotate every 180 days - RSA master keys: Rotate on compromise or annually

Access Control:

```
Lambda functions: Full access
Application servers: Read-only access (specific secrets)
Administrators: Read-only access (audit logged)
```

1.9.3 8.3 Logging and Monitoring

1.9.3.1 Security Events to Log

Event	Details	Retention
User Login	Username, timestamp, IP address, result	90 days
Failed Login	Username, timestamp, IP address, attempt count	90 days
File Upload	User ID, filename, size, encryption status	1 year
File Decryption	User ID, file ID, timestamp, success/failure	1 year
Authorization Failure	User ID, resource, attempted action	90 days
Configuration Change	Admin, change details, timestamp	1 year
Error Events	Error type, frequency, stack trace (sanitized)	30 days
System Events	Startup, shutdown, health check failures	90 days

Monitoring Rules:

```
Alert: Multiple failed login attempts
  Trigger: > 5 failed logins in 15 minutes
  Response: Temporary account lockout, admin notification

Alert: Unauthorized access attempts
  Trigger: Authorization failure logged
```

```
  Response: Log analysis, potential intrusion investigation

Alert: System health degradation
  Trigger: Response time > 5 seconds (99th percentile)
  Response: Auto-scaling triggered, admin notification

Alert: Configuration changes outside change control
  Trigger: Configuration modification without audit entry
  Response: Immediate admin notification, rollback consideration
```

1.9.4 8.4 Incident Response - Cryptographic Key Compromise

1.9.4.1 Scenario: Private Master Key Stolen Phase 1: Immediate Response (0-1 hour)

1. Confirm compromise
   - Review access logs for unauthorized access
   - Check key usage patterns
   - Validate integrity of key material
2. Contain incident
   - Immediately revoke compromised key in KMS
   - Disable service if necessary to prevent data exfiltration
   - Notify incident response team
3. Initial assessment
   - How much data was potentially encrypted with compromised key
   - When was key compromise likely (access logs)
   - Who had access to key

Action Plan:

```
# 1. Disable compromised key
aws kms disable-key --key-id <key-id>

# 2. Stop all decryption operations using this key
# (depends on architecture - may require service restart)
systemctl restart rsa_service

# 3. Rotate to new key
NEW_KEY_ID=$(aws kms create-key --description "RSA Master Key Rotated")
aws kms create-alias --alias-name alias/rsa-master-key-new --target-key-id $NEW_KEY_ID

# 4. Update application configuration
# (secret in Secrets Manager)
```

Communications: - Incident commander declares SEV-1 incident - Notify: CEO, Legal, PR, Customers - Template: “We identified potential unauthorized access to system. No evidence of data exfiltration yet.”

Phase 2: Short-term Response (1-24 hours)

1. Re-encrypt all data with new key

```
def rotate_encryption_keys():
    """Re-encrypt all files with new key"""
    conn = sqlite3.connect(DATABASE)
    cursor = conn.cursor()

    # Get all encrypted files
    cursor.execute('SELECT id, user_id, encrypted_data FROM files')
    files = cursor.fetchall()

    for file_id, user_id, old_encrypted_data in files:
        # Decrypt with old key (during key compromise window)
        plaintext = decrypt_with_old_key(old_encrypted_data)

        # Get user's public key
        cursor.execute('SELECT public_key FROM users WHERE id = ?', (user_id,))
        public_key_pem = cursor.fetchone()[0]

        # Re-encrypt with new key
        new_encrypted_data = encrypt_file(plaintext, public_key_pem)

        # Update database
        cursor.execute(
            'UPDATE files SET encrypted_data = ? WHERE id = ?',
            (new_encrypted_data, file_id)
        )

    conn.commit()
    conn.close()
```

2. Notify affected users
   - Email all users
   - Recommend password change
   - Recommend private key re-generation (if downloaded)
3. Forensics
   - Collect all logs for analysis
   - Identify who accessed key
   - Determine if data was accessed

Phase 3: Long-term Response (days-weeks)

1. Root cause analysis
   - How was key accessed (misconfiguration, human error, vulnerability)
   - What controls failed
   - Detailed timeline of events

2. Implement preventive controls
    - Hardware security module (HSM) for key storage
    - Multi-person authorization for key access
    - Encrypted logs with separate key management
    - Regular key access audits
3. Regulatory notifications
    - GDPR: If personal data exposed, notify authorities within 72 hours
    - HIPAA: Breach notification procedures
    - PCI-DSS: Incident reporting requirements
4. Customer communications
    - Explanation of what happened
    - Data protection measures implemented
    - Customer actions recommended
    - Ongoing monitoring commitment

Post-Incident Improvements:

1. Enhanced Key Management
    - Implement hardware security module (HSM)
    - Centralized key management service
    - Separation of duties (key generation, approval, usage)
2. Enhanced Monitoring
    - Real-time alerting on key access
    - Behavioral analysis of user/admin activities
    - Automated response to suspicious patterns
3. Enhanced Disaster Recovery
    - Regular backup and restore testing
    - Rapid key rotation procedures
    - Communication templates for incident notification

1.10 9. Maintenance, Evolution, and Cryptographic Agility

1.10.1 9.1 Post-Deployment SSDLC

Security engineering continues after deployment:

Continuous Monitoring: - Intrusion detection system (IDS) analysis - Log analysis for anomalies - Performance metrics indicating attacks - Security event aggregation and correlation

Vulnerability Management: - Regular vulnerability assessments - Prompt patching of identified issues - Security updates to dependencies - Configuration audits

Security Updates Process:

```
1. Vulnerability identified (internal or external report)
2. Severity assessment (CVSS scoring)
3. Impact analysis (affected services, data)
4. Fix development and testing
5. Change control approval
6. Staged deployment (dev -> staging -> prod)
7. Verification and monitoring
8. Post-deployment review
```

1.10.2 9.2 Cryptographic Agility

Definition: System's ability to quickly transition between cryptographic algorithms as threats evolve.

Current Threats to Cryptography:

1. **Post-Quantum Computing**
   - Timeline: Cryptographically relevant quantum computer estimated 2030+
   - Threat: RSA, ECC algorithms become obsolete
   - Impact: All encrypted data becomes vulnerable
2. **Hardware Vulnerabilities**
   - Spectre/Meltdown: Side-channel attacks on CPU
   - Impact: Potential key extraction from memory
3. **Algorithm Weaknesses**
   - New attacks on SHA-1, older AES implementations
   - Impact: Downgrade to weaker security

Agility Implementation:

1.10.2.1 Configuration-Driven Cryptography

```
# crypto_config.json
{
  "algorithms": {
    "asymmetric": {
      "current": "RSA",
      "config": {
        "key_size": 2048,
        "padding": "OAEP",
        "hash": "SHA256"
      }
    },
    "symmetric": {
      "current": "AES",
      "config": {
        "key_size": 256,
        "mode": "GCM",
```

```
        "iv_size": 128
      }
    },
    "hash": {
      "current": "SHA256"
    }
  },
  "transitions": {
    "post_quantum": {
      "target_algorithm": "Kyber",
      "migration_start": "2027-01-01",
      "cutover_date": "2029-01-01"
    }
  }
}

# Application uses configuration
config = json.load(open('crypto_config.json'))
algorithm = config['algorithms']['asymmetric']['current']
```

1.10.2.2 Post-Quantum Cryptography Migration Roadmap Phase 1: Assessment and Research (2027-2028)

- Evaluate NIST post-quantum candidates
- Assess algorithm maturity and performance
- Determine migration strategy for current deployments
- Action: Use Kyber-512 for pilot projects

Phase 2: Preparation (2027-2028)

- Implement hybrid encryption (RSA + Kyber)
- Deploy cryptographic agility framework
- Infrastructure for key generation and storage
- Action: All new deployments hybrid, existing TBD

Phase 3: Migration (2028-2029)

- Re-encrypt all data using post-quantum algorithms
- Transition production systems gradually
- Maintain backward compatibility period
- Action: Cut over to post-quantum as primary algorithm

Phase 4: Completion (2029-2030)

- Fully deprecate RSA
- Complete infrastructure transition
- Long-term support for post-quantum algorithms
- Action: Monitor for new quantum threats

Hybrid Encryption Example (current + post-quantum):

```
def encrypt_hybrid(file_data, public_key_rsa, public_key_pq):
    """Encrypt with both RSA and post-quantum algorithm"""

    # Generate symmetric key for data
    aes_key = os.urandom(32)

    # Encrypt symmetric key with both asymmetric algorithms
    encrypted_with_rsa = rsa_encrypt(aes_key, public_key_rsa)
    encrypted_with_pq = pq_encrypt(aes_key, public_key_pq)

    # Encrypt data with symmetric key
    encrypted_data = aes_encrypt(file_data, aes_key)

    # Return: [encrypted_with_rsa, encrypted_with_pq, encrypted_data]
    return package(encrypted_with_rsa, encrypted_with_pq, encrypted_data)

def decrypt_hybrid(encrypted_package, private_key_rsa, private_key_pq):
    """Decrypt using available key (RSA or post-quantum)"""
    encrypted_with_rsa, encrypted_with_pq, encrypted_data = unpackage(encrypted_package)

    try:
        # Try RSA first (currently most reliable)
        aes_key = rsa_decrypt(encrypted_with_rsa, private_key_rsa)
    except:
        try:
            # Fall back to post-quantum if RSA fails
            aes_key = pq_decrypt(encrypted_with_pq, private_key_pq)
        except:
            raise DecryptionFailure("Both RSA and post-quantum decryption failed")

    # Decrypt data
    plaintext = aes_decrypt(encrypted_data, aes_key)
    return plaintext
```

1.10.3 9.3 Long-Term Security Risks

1.10.3.1 Risk 1: Hard-Coded Cryptographic Choices

Problem: Algorithms selected at design time become ossified in code

```
# Hard-coded at design time
cipher_algorithm = "AES-128"  # Now considered weak

# Years later, changing requires code modification + deployment
```

Mitigation: - Use configuration files (JSON, YAML) - Support multiple algo-

rithms simultaneously - Parameterize cryptographic operations

1.10.3.2 Risk 2: Cryptocurrency Algorithm Deprecation

Problem: Standards bodies retire algorithms as attacks improve

Example: SHA-1 deprecated for digital signatures (collision attacks found)

Mitigation: - Use current standard algorithms (SHA-256+, RSA 2048+) - Plan algorithm transitions in advance - Monitor cryptographic research for vulnerabilities

1.10.3.3 Risk 3: Backward Compatibility Burden

Problem: Supporting old algorithms indefinitely creates security debt

```
# System must support old weak algorithm for compatibility
# But old algorithm is target for attacks
```

Mitigation: - Plan algorithm sunset dates - Communicate deprecation clearly - Force re-encryption before sunset - Accept temporary service disruption

1.10.3.4 Risk 4: Regulatory Changes

Problem: New regulations may mandate specific cryptographic practices

Example: EU regulations may require EU-approved algorithms only

Mitigation: - Monitor regulatory landscape - Implement compliance checking - Design for regulatory flexibility - Maintain compliance documentation

---

1.11 10. Conclusion

This comprehensive analysis demonstrates the security engineering required to develop a cryptographic service following SSDLC principles. Key findings:

1.11.1 Security Achievements

1. **Requirements Engineering**: Mapped 16 security requirements to OWASP ASVS, NIST CSF, ISO/IEC 27001 standards
2. **Threat Analysis**: Identified 24 STRIDE threats + 5 detailed scenarios with risk assessment
3. **Architecture**: Multi-tier design with defense-in-depth, least privilege, fail-secure principles
4. **Cryptography**: OAEP padding, hybrid encryption, Encrypt-then-MAC integrity protection
5. **Authentication**: Multi-factor authentication support, JWT token-based sessions
6. **Authorization**: Role-based access control, horizontal privilege escalation prevention

7. **Audit**: Comprehensive logging of all security-relevant events
8. **Testing**: SAST tools, unit testing, integration testing, security-specific tests
9. **Deployment**: Cloud VPC architecture, HSM integration, secret management
10. **Operations**: Incident response procedures, key compromise handling, disaster recovery

1.11.2 Intentional Vulnerabilities for Learning

The provided code includes 12 documented vulnerabilities:

1. Hard-coded SECRET_KEY
2. Weak password requirements
3. PKCS#1 v1.5 padding (padding oracle)
4. Direct RSA encryption (size limitation)
5. No file size validation
6. Missing authorization checks
7. Private key exposure to client
8. Missing integrity protection
9. Incomplete audit logging
10. Debug mode enabled
11. Weak input validation
12. No rate limiting

Each vulnerability includes explanation, attack scenario, and secure implementation, enabling students to understand real-world security flaws and defenses.

1.11.3 Post-Quantum Considerations

The document addresses emerging threats from quantum computing: - 4-phase migration roadmap (2027-2030) - Hybrid encryption approach (RSA + Kyber) - Configuration-driven algorithm selection - Backward compatibility strategies

1.11.4 Continuous Improvement

Security is not a one-time achievement but continuous evolution: - Regular vulnerability assessments - Security updates process - Incident response procedures - Compliance monitoring - Cryptographic agility

1.11.5 Recommendation

Organizations implementing cryptographic services should:

1. Establish SSDLC governance (not ad-hoc security)
2. Conduct threat modeling early in design
3. Use validated cryptographic libraries (not homegrown)
4. Implement comprehensive audit logging

5. Plan for cryptographic evolution
6. Regular security testing and assessment
7. Security training for development teams
8. Incident response procedures before needed
9. Compliance mapping to relevant standards
10. Continuous monitoring and improvement

The RSA encryption service, while simplified for educational purposes, demonstrates that security-critical systems require sophisticated engineering across requirements, design, implementation, testing, deployment, and operations phases. No single control is sufficient - defense-in-depth requires multiple layers of security working together.

1.12 References

1. OWASP. (2023). Application Security Verification Standard (ASVS). https://owasp.org/www-project-application-security-verification-standard/
2. NIST. (2022). Cybersecurity Framework. https://www.nist.gov/cyberframework
3. ISO/IEC. (2013). Information security management systems. ISO/IEC 27001:2013
4. Microsoft. (2023). Security Development Lifecycle. https://www.microsoft.com/en-us/securityengineering/sdl/
5. Cryptography Research, Inc. (2023). Cryptographic Best Practices. https://crypto.stackexchange.com
6. OWASP. (2023). Top 10 Web Application Security Risks. https://owasp.org/Top10/
7. NIST. (2022). Post-Quantum Cryptography Standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/
8. Google. (2023). Secure Software Supply Chain. https://cloud.google.com/architecture/devops-culture-supply-chain-security
9. Schneier, B. (2015). Applied Cryptography (2nd ed.). John Wiley & Sons.
10. McGraw, G. (2006). Software Security: Building Security In. Addison-Wesley Professional.