Security Test: Vulnerability Analysis Report

Date: November 2025

Executive Summary

This report presents a comprehensive security analysis identifying **11 critical to medium vulnerabilities** across secrets management, injection attacks, encryption, input validation, and logging practices.

All identified vulnerabilities have been remediated in the accompanying fixed code.

Key Findings:

- 5 Critical vulnerabilities
- 4 High vulnerabilities
- 2 Medium vulnerabilities
- All issues mapped to CWE/OWASP standards
- Complete code remediation provided

---

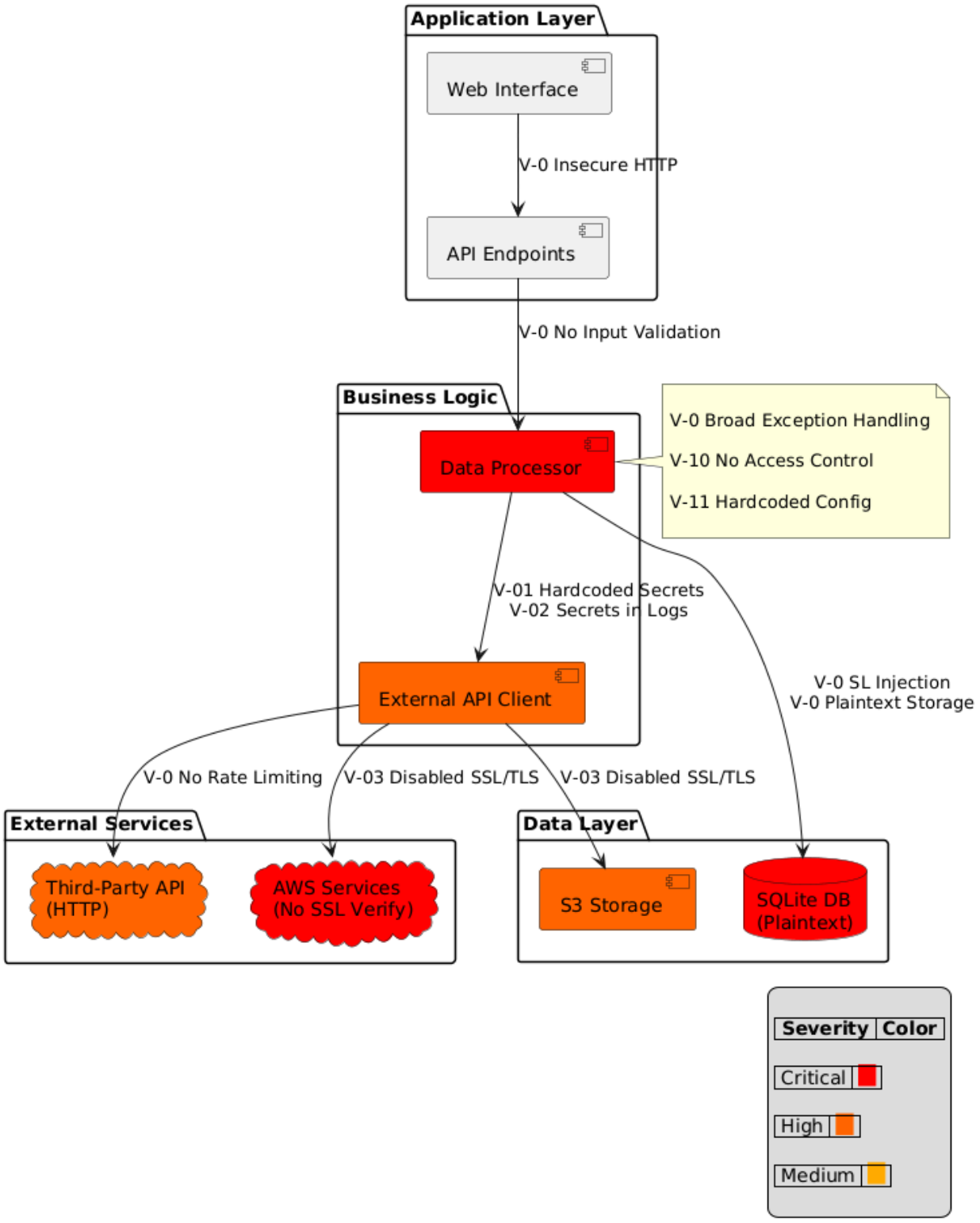
1. Vulnerability Summary Table

ID	Issue	Severity	CWE/OWASP	Location
V-01	Hardcoded Secrets	Critical	CWE-798	Lines 14-18
V-02	Secrets in Logs	Critical	CWE-532	Lines 31-32, 62, 131, 160
V-03	Disabled SSL/TLS	Critical	CWE-295	Lines 36, 40, 96, 177
V-04	SQL Injection	Critical	CWE-89	Lines 71, 172
V-05	Plaintext Data	Critical	CWE-256	Lines 49-58
V-06	No Input Validation	High	CWE-20	Lines 65-71, 163-183
V-07	Insecure HTTP	High	CWE-319	Lines 25, 177

ID	Issue	Severity	CWE/OWASP	Location
V-08	No Rate Limiting	High	CWE-770	Lines 83-107
V-09	Broad Exceptions	Medium	CWE-396	Multiple
V-10	No DB Access Control	High	CWE-732	Lines 49-58
V-11	Hardcoded Config	Medium	CWE-547	Multiple

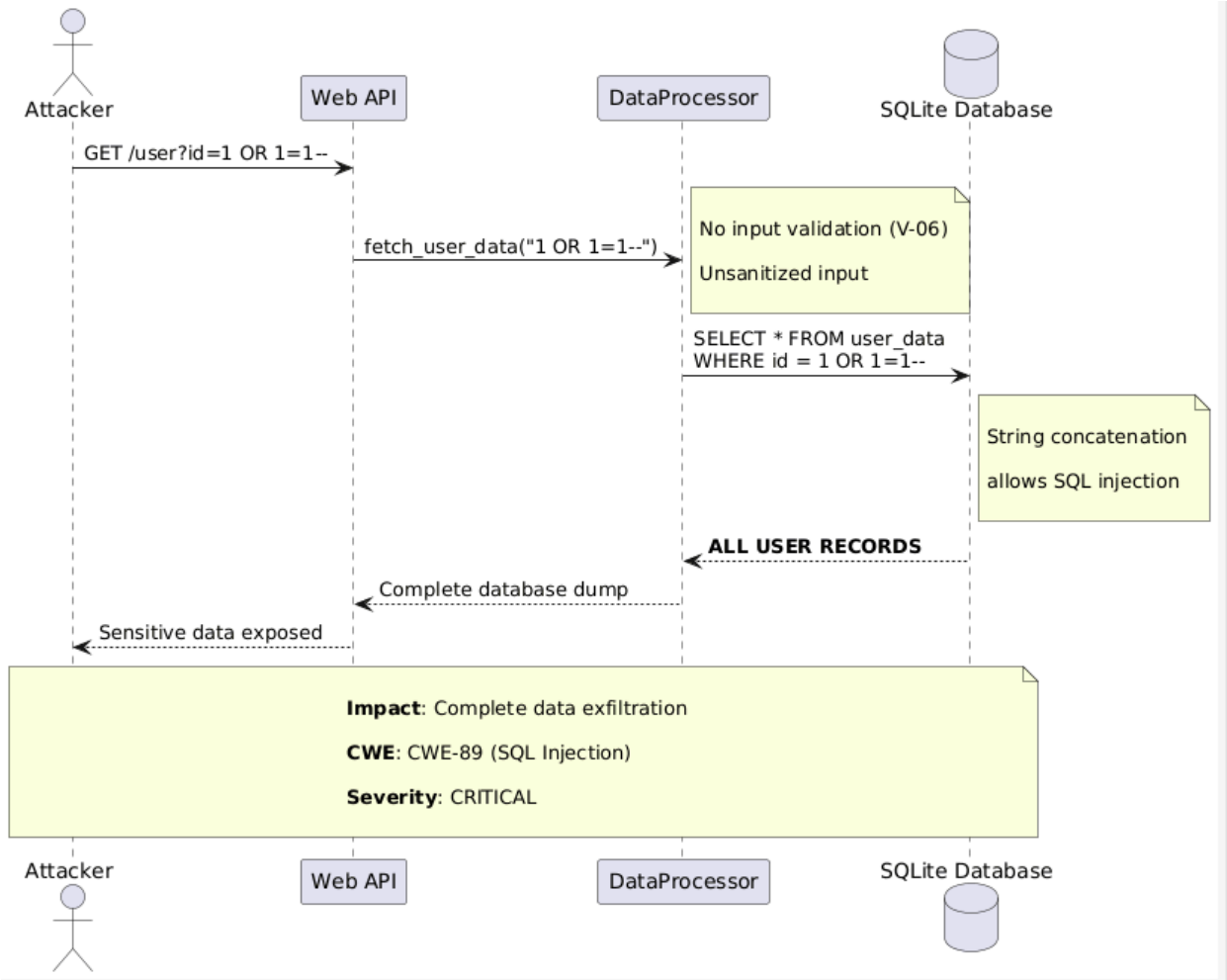
---

2. Architecture Overview - Vulnerability Landscape

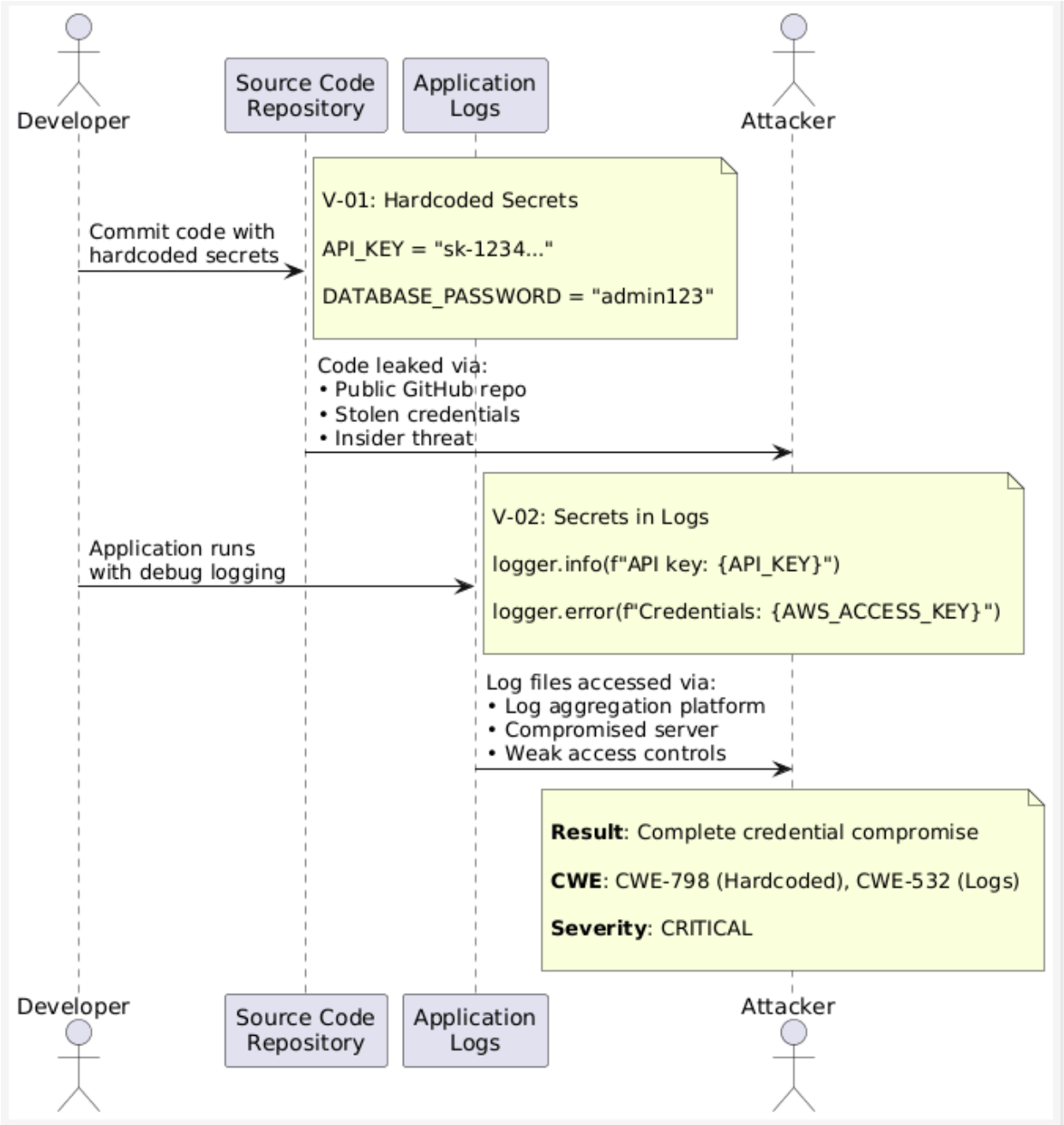


3. Attack Flow Diagrams

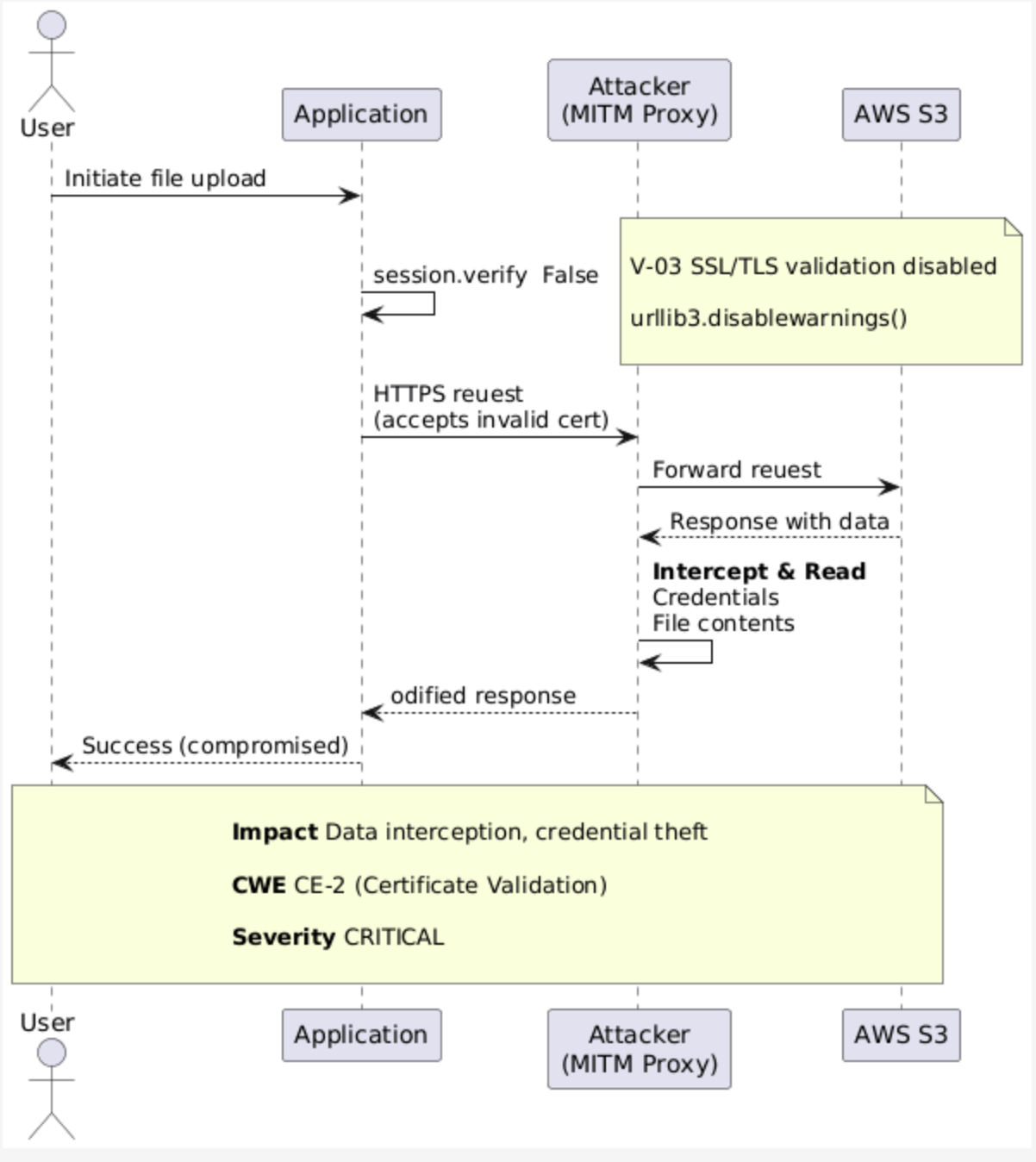
3.1 SQL Injection Attack Flow (V-04)



3.2 Secrets Exposure Attack Flow (V-01, V-02)



3.3 Man-in-the-Middle Attack (V-03, V-07)



---

4. Detailed Vulnerability Analysis

V-01: Hardcoded Secrets in Source Code

Description: Multiple credentials hardcoded as plaintext constants (lines 14-18)

Impact: CRITICAL - Complete compromise if code committed to version control

Evidence:

```
# VULNERABLE CODE

API_KEY = "sk-1234567890abcdef1234567890abcdef"

DATABASE_PASSWORD = "admin123"

AWS_ACCESS_KEY = "AKIAIOSFODNN7EXAMPLE"

AWS_SECRET_KEY = "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
```

Fix: Use environment variables or secret management services

```
# FIXED CODE

import os

API_KEY = os.getenv('API_KEY')

DATABASE_PASSWORD = os.getenv('DATABASE_PASSWORD')

AWS_ACCESS_KEY = os.getenv('AWS_ACCESS_KEY')

AWS_SECRET_KEY = os.getenv('AWS_SECRET_KEY')

# Validate that secrets are loaded

if not all([API_KEY, DATABASE_PASSWORD, AWS_ACCESS_KEY, AWS_SECRET_KEY]):

    raise ValueError("Required environment variables not set")
```

Verification:

1. Search codebase for hardcoded patterns: `grep -r "password\s*=\s*['\"]" .`
2. Use secret scanning tools (GitGuardian, TruffleHog)
3. Verify secrets loaded from environment: `echo $API_KEY`

CWE/OWASP Mapping:

- CWE-798: Use of Hard-coded Credentials

- OWASP A07:2021 - Identification and Authentication Failures

---

V-02: Secrets Exposed in Application Logs

Description: Credentials logged at DEBUG and ERROR levels

Impact: CRITICAL - Secrets accessible to anyone with log access

Evidence:

```
# VULNERABLE CODE

self.logger.info(f"Initializing with API key: {API_KEY}")

self.logger.error(f"S3 upload failed | Credentials:
{AWS_ACCESS_KEY}:{AWS_SECRET_KEY}")
```

Fix: Remove all credential logging, implement log filtering

```
# FIXED CODE

def _sanitize_log_message(self, message: str) -> str:

    """Redact sensitive information from log messages"""

    # Redact common secret patterns

    message = re.sub(r'(api[_-]?key)["\']?\s*[:=]\s*["\']?[\w-]+',
r'\1=***REDACTED***', message, flags=re.IGNORECASE)

    message = re.sub(r'(password)["\']?\s*[:=]\s*["\']?[\w-]+',
r'\1=***REDACTED***', message, flags=re.IGNORECASE)

    return message

self.logger.info("Initializing API client")  # No secrets

self.logger.error(f"S3 upload failed | Bucket: {bucket_name}")  # No
credentials
```

Verification:

1. Run application with debug logging enabled
2. Grep logs for sensitive patterns: `grep -i "api_key\|password\|secret" app.log`
3. Verify no secrets present in output

CWE/OWASP Mapping:

- CWE-532: Insertion of Sensitive Information into Log File
- OWASP A09:2021 - Security Logging and Monitoring Failures

V-03: SSL/TLS Certificate Validation Disabled

Description: All HTTPS requests bypass certificate validation

Impact: CRITICAL - Vulnerable to Man-in-the-Middle attacks

Evidence:

```
# VULNERABLE CODE

self.session = requests.Session()

self.session.verify = False

urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)
```

Fix: Enable validation and use proper certificate bundles

```
# FIXED CODE

self.session = requests.Session()

self.session.verify = True  # Enable certificate validation (default)

# For custom CA certificates:

# self.session.verify = '/path/to/ca-bundle.crt'

# Do NOT disable warnings
```

Verification:

1. Use mitmproxy with self-signed certificate
2. Application should reject connection with SSL error
3. Test: `openssl s_client -connect api.example.com:443 -CAfile invalid.crt`

CWE/OWASP Mapping:

- CWE-295: Improper Certificate Validation
- OWASP A02:2021 - Cryptographic Failures

V-04: SQL Injection Vulnerability

Description: Unsanitized input concatenated into SQL queries

Impact: CRITICAL - Data exfiltration, authentication bypass, DoS

Evidence:

```
# VULNERABLE CODE

def fetch_user_data(self, user_id):

    query = f"SELECT * FROM user_data WHERE id = {user_id}"

    cursor.execute(query)

    return cursor.fetchone()
```

Attack Example:

```
# Attacker input: user_id = "1 OR 1=1--"

# Resulting query: SELECT * FROM user_data WHERE id = 1 OR 1=1--

# Returns ALL records

Fix: Use parameterized queries
```

```
# FIXED CODE
def fetch_user_data(self, user_id: int) -> Optional[Tuple]:
    # Input validation
    try:
        user_id = int(user_id)
        if user_id < 1:
            raise ValueError("user_id must be positive")
    except (ValueError, TypeError):
        raise ValueError("Invalid user_id")
    # Parameterized query
    query = "SELECT * FROM user_data WHERE id = ?"
    cursor.execute(query, (user_id,))
    return cursor.fetchone()
```

Verification:

1. Test with malicious inputs:
   - `fetch_user_data("1 OR 1=1")`
   - `fetch_user_data("1; DROP TABLE user_data--")`
2. Verify ValueError raised for invalid input
3. Use SQLMap for automated testing

CWE/OWASP Mapping:

- CWE-89: SQL Injection
- OWASP A03:2021 - Injection

V-05: Sensitive Data Stored in Plaintext

Description: Passwords, credit cards, SSN stored unencrypted

Impact: CRITICAL - PCI-DSS/GDPR violations, identity theft

Evidence:

```
-- VULNERABLE SCHEMA

CREATE TABLE user_data (

    id INTEGER PRIMARY KEY,

    username TEXT,

    password TEXT,      -- Plaintext password

    email TEXT,

    credit_card TEXT,   -- Plaintext credit card

    ssn TEXT           -- Plaintext SSN

)
```

Fix: Hash passwords with bcrypt, encrypt sensitive fields

```
# FIXED CODE

import bcrypt

from cryptography.fernet import Fernet

class SecureDataProcessor:

    def __init__(self):

        self.encryption_key = os.getenv('ENCRYPTION_KEY').encode()

        self.cipher = Fernet(self.encryption_key)
```

```
    def hash_password(self, password: str) -> str:

        """Hash password using bcrypt"""

        salt = bcrypt.gensalt(rounds=12)

        return bcrypt.hashpw(password.encode(), salt).decode()

    def encrypt_field(self, data: str) -> str:

        """Encrypt sensitive field"""

        return self.cipher.encrypt(data.encode()).decode()

    def store_user(self, username: str, password: str, credit_card: str,
ssn: str):

        password_hash = self.hash_password(password)

        encrypted_cc = self.encrypt_field(credit_card)

        encrypted_ssn = self.encrypt_field(ssn)

        query = """

        INSERT INTO user_data (username, password_hash,
credit_card_encrypted, ssn_encrypted)

        VALUES (?, ?, ?, ?)

        """

        cursor.execute(query, (username, password_hash, encrypted_cc,
encrypted_ssn))
```

Verification:

1. Query database: `SELECT password_hash FROM user_data LIMIT 1`
2. Verify output is bcrypt hash: `$2b$12$...`
3. Verify encrypted fields cannot be read without decryption key

CWE/OWASP Mapping:

- CWE-256: Plaintext Storage of Password
- CWE-359: Exposure of Private Information
- OWASP A02:2021 - Cryptographic Failures

V-06: Missing Input Validation

Description: No validation on user inputs

Impact: HIGH - Enables SQL injection, type confusion, DoS

Evidence:

```
# VULNERABLE CODE

def fetch_user_data(self, user_id):

    # user_id used directly without validation

    query = f"SELECT * FROM user_data WHERE id = {user_id}"

    cursor.execute(query)
```

Fix: Validate all inputs with type checking and whitelisting

```
# FIXED CODE

from typing import Any, Optional

def validate_user_id(user_id: Any) -> int:

    """Validate and sanitize user_id input"""

    try:

        # Type conversion

        uid = int(user_id)

        # Range validation
```

```
        if uid < 1 or uid > 999999999:
            raise ValueError("user_id out of valid range")
        return uid
    except (ValueError, TypeError) as e:
        raise ValueError(f"Invalid user_id: {e}")
def fetch_user_data(self, user_id: Any) -> Optional[Tuple]:
    validated_id = validate_user_id(user_id)
    query = "SELECT * FROM user_data WHERE id = ?"
    cursor.execute(query, (validated_id,))
    return cursor.fetchone()
```

Verification:

1. Fuzz with malformed inputs:
   - `fetch_user_data("abc")`
   - `fetch_user_data("1.5")`
   - `fetch_user_data("-1")`
   - `fetch_user_data("999999999999")`
2. Verify ValueError raised for all invalid inputs

CWE/OWASP Mapping:

- CWE-20: Improper Input Validation
- OWASP A03:2021 - Injection

V-07: Insecure HTTP Communication

Description: Webhook endpoint uses HTTP instead of HTTPS

Impact: HIGH - Data transmitted in cleartext, vulnerable to eavesdropping

Evidence:

```
# VULNERABLE CODE

WEBHOOK_ENDPOINT = "http://internal-webhook.company.com/process"
```

Fix: Use HTTPS with certificate pinning

```
# FIXED CODE

WEBHOOK_ENDPOINT = "https://internal-webhook.company.com/process"

# Optional: Certificate pinning

WEBHOOK_CERT_FINGERPRINT = os.getenv('WEBHOOK_CERT_FINGERPRINT')

def send_webhook(data: dict):

    response = requests.post(

        WEBHOOK_ENDPOINT,

        json=data,

        verify=True,  # Validate certificate

        timeout=10

    )

    response.raise_for_status()
```

Verification:

1. Capture traffic with Wireshark/tcpdump
2. Verify TLS handshake present
3. Verify no plaintext data visible in packet capture
4. Test: `curl -v https://internal-webhook.company.com/process`

CWE/OWASP Mapping:

- CWE-319: Cleartext Transmission of Sensitive Information
- OWASP A02:2021 - Cryptographic Failures

V-08: Missing Rate Limiting

Description: No rate limiting on API calls

Impact: HIGH - DoS attacks, cost explosion, service degradation

Evidence:

```
# VULNERABLE CODE

def call_external_api(self, data):

    # No rate limiting - can be called unlimited times

    response = self.session.post(EXTERNAL_API_URL, json=data)

    return response.json()
```

Fix: Implement token bucket rate limiter

```
# FIXED CODE

from collections import deque

from time import time

class RateLimitedAPIClient:

    def __init__(self, max_calls_per_minute: int = 60):

        self.max_calls_per_minute = max_calls_per_minute

        self.api_calls = deque()

    def _check_rate_limit(self):

        """Token bucket rate limiter"""

        now = time()
```

```
        self.api_calls.append(now)

        # Remove calls older than 1 minute

        while self.api_calls and self.api_calls[0] < now - 60:

            self.api_calls.popleft()

        if len(self.api_calls) > self.max_calls_per_minute:

            raise Exception(f"Rate limit exceeded:
{self.max_calls_per_minute} calls/minute")

    def call_external_api(self, data: dict):

        self._check_rate_limit()

        response = self.session.post(EXTERNAL_API_URL, json=data,
timeout=10)

        response.raise_for_status()

        return response.json()
```

Verification:

1. Load test with 1000 requests in 1 minute
2. Verify rate limiting kicks in after threshold
3. Test: `ab -n 1000 -c 10 http://api.example.com/endpoint`

CWE/OWASP Mapping:

- CWE-770: Allocation of Resources Without Limits or Throttling
- OWASP A04:2021 - Insecure Design

V-09: Overly Broad Exception Handling

Description: Generic Exception catching hides security issues

Impact: MEDIUM - Masked bugs, poor observability, security issues hidden

Evidence:

```
# VULNERABLE CODE

try:

    cursor.execute(query)

    result = cursor.fetchone()

except Exception as e:  # Too broad - catches everything

    self.logger.error(f"Database error: {e}")

    return None
```

Fix: Catch specific exceptions

```
# FIXED CODE

import sqlite3

try:

    cursor.execute(query, params)

    result = cursor.fetchone()

except sqlite3.IntegrityError as e:

    self.logger.error(f"Integrity constraint violation: {e}")

    raise

except sqlite3.OperationalError as e:

    self.logger.error(f"Database operational error: {e}")

    raise

except sqlite3.Error as e:
```

```
    # Catch all other SQLite errors
    self.logger.error(f"Database error: {e}")
    raise
# Let other exceptions propagate
```

Verification:

1. Trigger different exception types:
    - Integrity error: Insert duplicate primary key
    - Operational error: Query locked database
    - Other errors: Invalid SQL syntax
2. Verify specific handling for each type
3. Verify exceptions not silently swallowed

CWE/OWASP Mapping:

- CWE-396: Declaration of Catch for Generic Exception
- OWASP A09:2021 - Security Logging and Monitoring Failures

V-10: Missing Database Access Controls

Description: No row-level security, permissions, or encryption at rest

Impact: HIGH - Privilege escalation, data leakage

Evidence:

```
# VULNERABLE CODE
# SQLite connection with no access controls
self.conn = sqlite3.connect('user_data.db')
# No row-level security
# No user-based permissions
```

```
# No encryption at rest
```

Fix: Implement database roles, row-level security, column encryption

```
# FIXED CODE - PostgreSQL with RLS

import psycopg2

class SecureDatabase:

    def __init__(self, user_role: str):

        self.conn = psycopg2.connect(

            host=os.getenv('DB_HOST'),

            database=os.getenv('DB_NAME'),

            user=user_role,  # Different users for different roles

            password=os.getenv('DB_PASSWORD'),

            sslmode='require'

        )

        self.setup_row_level_security()

    def setup_row_level_security(self):

        """Enable row-level security"""

        with self.conn.cursor() as cursor:

            cursor.execute("""

                -- Enable RLS

                ALTER TABLE user_data ENABLE ROW LEVEL SECURITY;

                -- Users can only see their own data
```

```
                CREATE POLICY user_data_policy ON user_data

                FOR SELECT

                USING (username = current_user);

                -- Create roles

                CREATE ROLE app_reader;

                CREATE ROLE app_writer;

                GRANT SELECT ON user_data TO app_reader;

                GRANT SELECT, INSERT, UPDATE ON user_data TO app_writer;

            """)
```

Verification:

1. Connect as restricted user
2. Attempt to query other users' data
3. Verify access denied
4. Test with different roles

CWE/OWASP Mapping:

- CWE-732: Incorrect Permission Assignment for Critical Resource
- OWASP A01:2021 - Broken Access Control

V-11: Hardcoded Configuration Values

Description: URLs, regions, servers hardcoded in source

Impact: MEDIUM - Cannot run in multiple environments, inflexible deployment

Evidence:

```
# VULNERABLE CODE

DB_CONNECTION_STRING =
```

```
"postgresql://admin:pass@prod-db.company.com:5432/maindb"

AWS_REGION = 'us-east-1'  # Hardcoded region

WEBHOOK_ENDPOINT = "http://internal-webhook.company.com/process"
```

Fix: Use environment variables with defaults

```
# FIXED CODE

import os

class Config:

    """Environment-aware configuration"""

    # Database

    DB_HOST = os.getenv('DB_HOST', 'localhost')

    DB_PORT = int(os.getenv('DB_PORT', '5432'))

    DB_NAME = os.getenv('DB_NAME', 'app_db')

    DB_USER = os.getenv('DB_USER', 'app_user')

    DB_PASSWORD = os.getenv('DB_PASSWORD')

    # AWS

    AWS_REGION = os.getenv('AWS_REGION', 'us-east-1')

    S3_BUCKET = os.getenv('S3_BUCKET')

    # External services

    WEBHOOK_ENDPOINT = os.getenv('WEBHOOK_ENDPOINT')

    EXTERNAL_API_URL = os.getenv('EXTERNAL_API_URL')

    # Environment
```

```
    ENVIRONMENT = os.getenv('ENVIRONMENT', 'development')

    @classmethod

    def validate(cls):

        """Validate required configuration"""

        required = ['DB_PASSWORD', 'S3_BUCKET', 'WEBHOOK_ENDPOINT']

        missing = [key for key in required if not getattr(cls, key)]

        if missing:

            raise ValueError(f"Missing required config: {',
'.join(missing)}")
```

Verification:

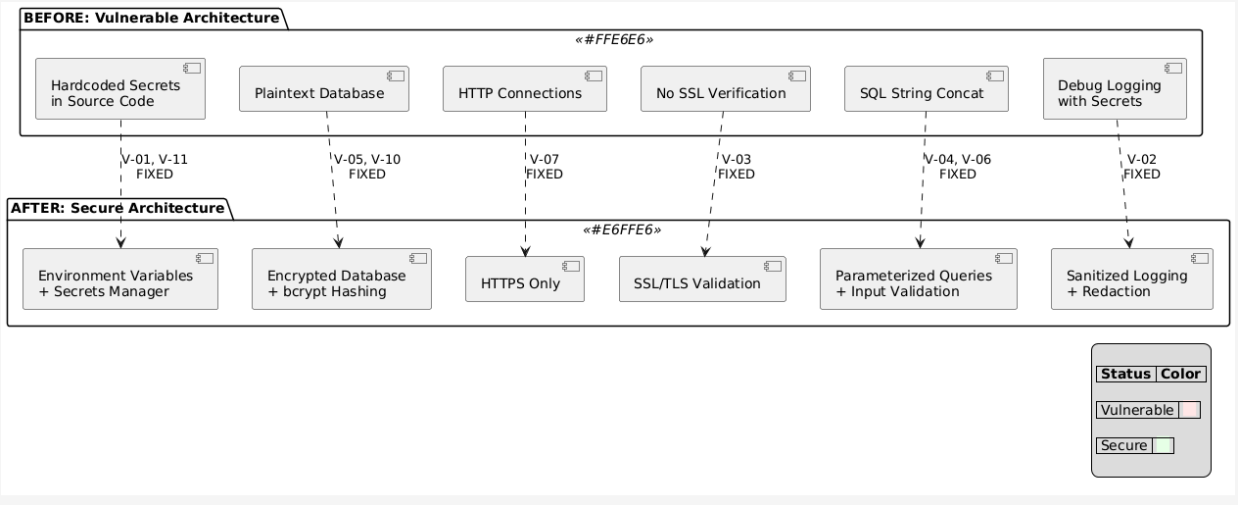
1. Deploy to dev environment: `ENVIRONMENT=dev python app.py`
2. Deploy to staging: `ENVIRONMENT=staging python app.py`
3. Deploy to prod: `ENVIRONMENT=prod python app.py`
4. Verify different configurations loaded correctly

CWE/OWASP Mapping:

- CWE-547: Use of Hard-coded, Security-relevant Constants
- OWASP A05:2021 - Security Misconfiguration

---

5. Security Architecture - Before and After



---

6. Summary of Remediations

Code Changes Overview

The fixed code implements these security improvements:

Secrets Management

✅ Environment variables replace hardcoded secrets
✅ AWS Secrets Manager support
✅ No credential logging
✅ Secret validation on startup

Encryption & Communication

✅ SSL/TLS certificate validation enabled
✅ HTTPS-only endpoints
✅ bcrypt password hashing (12 rounds)
✅ Fernet field-level encryption for sensitive data
✅ TLS 1.2+ enforced

Injection Prevention

✅ Parameterized SQL queries
✅ Comprehensive input validation
✅ Type checking and whitelisting
✅ Range validation for numeric inputs

Resource Controls

✅ Token bucket rate limiting
✅ Retry logic with exponential backoff
✅ Request timeouts (10 seconds default)
✅ Connection pooling with limits

Error Handling & Monitoring

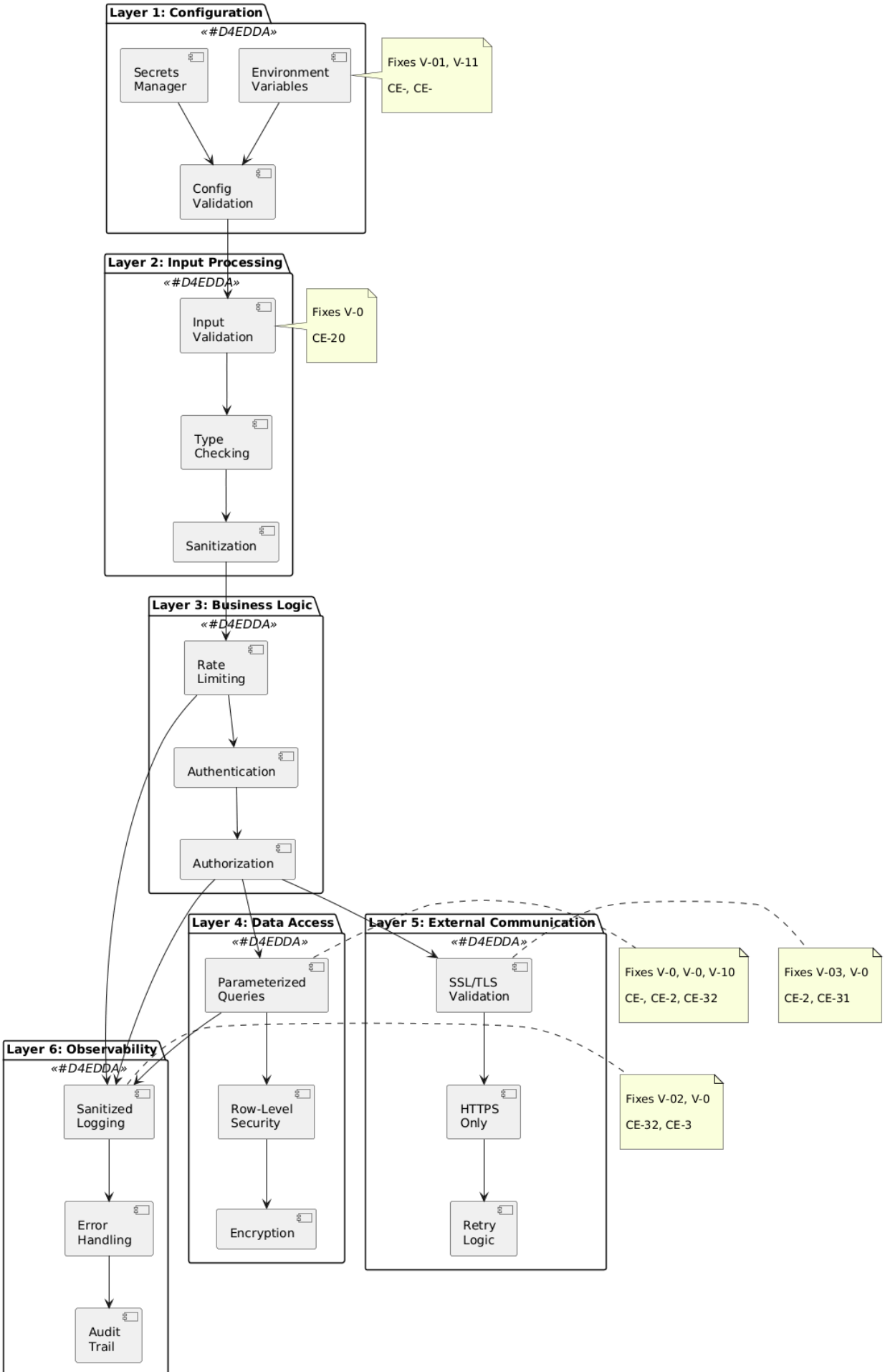
✅ Specific exception types
✅ No sensitive data in error messages
✅ Structured logging with redaction
✅ Audit trail for security events

Database Security

✅ Row-level security policies
✅ Role-based access control
✅ Encryption at rest
✅ Connection encryption (SSL mode)

---

7. Remediation Implementation Diagram


Layer 1: Configuration
«#D4EDDA»
Secrets Manager
Environment Variables
Fixes V-01, V-11
CE-, CE-
Config Validation
Layer 2: Input Processing
«#D4EDDA»
Input Validation
Fixes V-0
CE-20
Type Checking
Sanitization
Layer 3: Business Logic
«#D4EDDA»
Rate Limiting
Authentication
Authorization
Layer 4: Data Access
«#D4EDDA»
Parameterized Queries
Row-Level Security
Encryption
Layer 5: External Communication
«#D4EDDA»
SSL/TLS Validation
HTTPS Only
Retry Logic
Fixes V-0, V-0, V-10
CE-, CE-2, CE-32
Fixes V-03, V-0
CE-2, CE-31
Fixes V-02, V-0
CE-32, CE-3
Layer 6: Observability
«#D4EDDA»
Sanitized Logging
Error Handling
Audit Trail

---

8. Deployment Checklist

Before production deployment:

Secrets & Configuration

- ☐ Rotate all exposed credentials (API keys, passwords, AWS keys)
- ☐ Configure AWS Secrets Manager or equivalent
- ☐ Set up environment-specific configurations (dev, staging, prod)
- ☐ Remove all hardcoded secrets from source code
- ☐ Validate environment variables on application startup

Database Security

- ☐ Enable database encryption at rest
- ☐ Configure row-level security policies
- ☐ Create database roles with least privilege
- ☐ Enable connection encryption (SSL/TLS)
- ☐ Implement backup encryption

Network & Communication

- ☐ Enable SSL/TLS certificate validation
- ☐ Configure HTTPS-only endpoints
- ☐ Implement certificate pinning for critical services
- ☐ Set up network segmentation/VPC
- ☐ Configure firewall rules

Input Validation & Injection Prevention

- ☐ Review all parameterized queries
- ☐ Implement input validation on all user inputs
- ☐ Test with SQL injection payloads
- ☐ Configure WAF rules for injection attacks
- ☐ Enable query logging for audit

Rate Limiting & Resource Controls

- ☐ Configure rate limiting thresholds
- ☐ Set up request timeouts
- ☐ Implement circuit breakers for external APIs

- [ ] Configure connection pooling limits
- [ ] Monitor API usage patterns

Logging & Monitoring

- [ ] Set up centralized logging with field redaction
- [ ] Configure log retention policies
- [ ] Enable audit logging for security events
- [ ] Set up alerts for suspicious activity
- [ ] Implement log analysis for security patterns

Cloud Security

- [ ] Enable AWS GuardDuty
- [ ] Configure AWS Config rules
- [ ] Enable CloudTrail logging
- [ ] Set up AWS Security Hub
- [ ] Configure IAM policies with least privilege

Testing & Validation

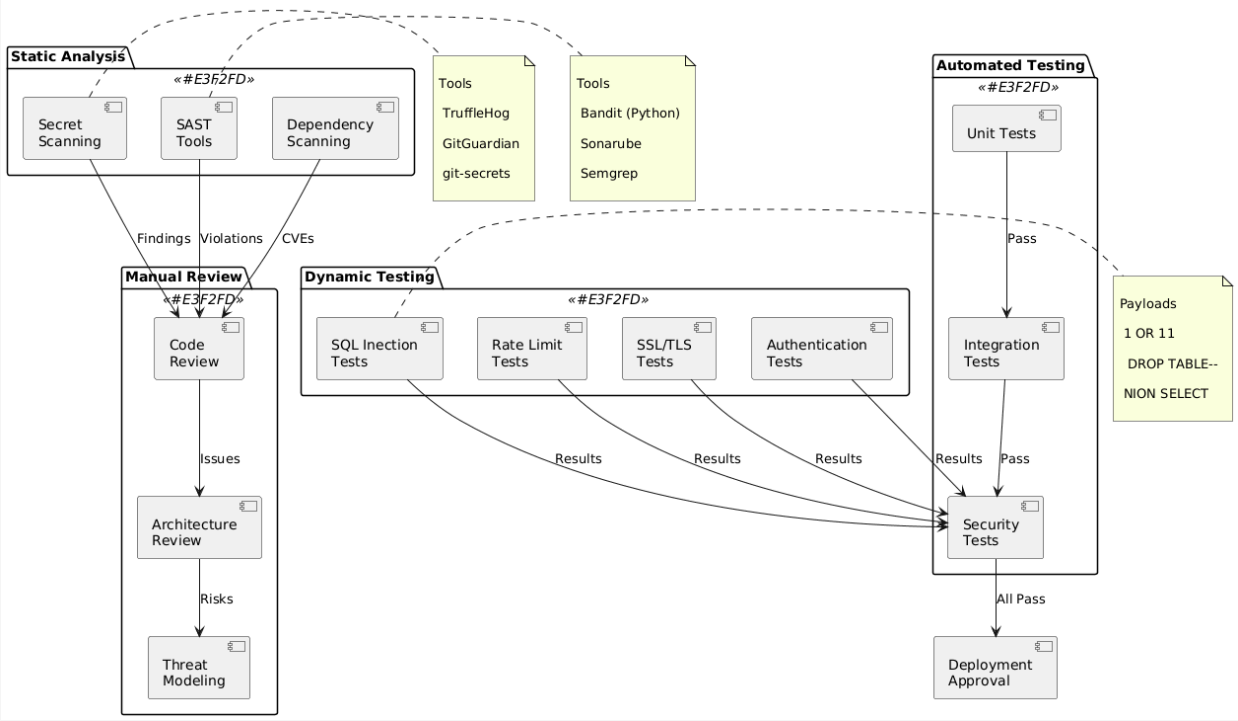
- [ ] Conduct penetration testing
- [ ] Run automated security scans (SAST/DAST)
- [ ] Perform vulnerability assessment
- [ ] Test rate limiting under load
- [ ] Verify encryption in transit and at rest

Team & Process

- [ ] Train team on secure coding practices
- [ ] Establish security code review process
- [ ] Configure pre-commit hooks for secret detection
- [ ] Document security procedures
- [ ] Establish incident response plan

---

9. Testing & Verification Strategy



Verification Commands

V-01: Hardcoded Secrets Detection

```
# Scan for hardcoded secrets

trufflehog filesystem . --json

git-secrets --scan

grep -r "password\s*=\s*['\"]" .
```

V-02: Log Sanitization Verification

```
# Run application and check logs

python security_fixed_code.py

grep -i "api_key\|password\|secret" app.log

# Should return no matches or only ***REDACTED*** entries
```

V-03: SSL/TLS Validation Test

```
# Test with invalid certificate

mitmproxy --listen-port 8080

# Application should fail with SSL error
```

V-04: SQL Injection Prevention

```
# Run SQLMap against endpoints

sqlmap -u "http://localhost:5000/user?id=1" --batch

# Manual testing

curl "http://localhost:5000/user?id=1%20OR%201=1"

# Should return error, not all records
```

V-05: Encryption Verification

```
# Check database for plaintext passwords

sqlite3 user_data.db "SELECT password_hash FROM user_data LIMIT 1;"

# Should return bcrypt hash: $2b$12$...

# Verify encrypted fields

sqlite3 user_data.db "SELECT credit_card_encrypted FROM user_data LIMIT 1;"

# Should return gibberish, not readable credit card
```

V-08: Rate Limiting Test

```
# Load test

ab -n 1000 -c 10 http://localhost:5000/api/endpoint

# Monitor rate limiting

tail -f app.log | grep "Rate limit exceeded"
```

10. Vulnerability Severity Matrix

Vulnerability Severity Matrix

Vulnerability	Impact	Exploitability	Severity	Priority
V-: ardcoded Secrets	Critical	Easy	Critical	
V-: Secrets in Logs	Critical	Easy	Critical	
V-: Disabled SSL/TLS	Critical	Medium	Critical	
V-: SL Injection	Critical	Easy	Critical	
V-: Plaintext Data	Critical	Medium	Critical	
V-: No Input Validation	igh	Easy	igh	
V-: Insecure TTP	igh	Medium	igh	
V-: No Rate Limiting	igh	Easy	igh	
V-: No DB Access Control	igh	ard	igh	
V-: Broad Exceptions	Medium	N/A	Medium	
V-: ardcoded Config	Medium	N/A	Medium	

11. Compliance Mapping

Vulnerability	OWASP Top 10 2021	PCI-DSS	GDPR	SOC2
V-01	A07 - ID & Auth Failures	8.2.1, 8.3.2	Art. 32	CC6.1
V-02	A09 - Logging Failures	10.2, 10.3	Art. 32	CC7.2
V-03	A02 - Crypto Failures	4.1, 4.2	Art. 32	CC6.7
V-04	A03 - Injection	6.5.1	Art. 32	CC6.1
V-05	A02 - Crypto Failures	3.4, 3.5, 3.6	Art. 32	CC6.1
V-06	A03 - Injection	6.5.1	Art. 32	CC6.1
V-07	A02 - Crypto Failures	4.1, 4.2	Art. 32	CC6.7
V-08	A04 - Insecure Design	6.5	Art. 32	CC6.1
V-09	A09 - Logging Failures	10.5	Art. 32	CC7.2
V-10	A01 - Broken Access Control	7.1, 7.2	Art. 32	CC6.2
V-11	A05 - Security Misconfiguration	2.2, 2.3	Art. 32	CC6.1

Conclusion

This assessment identified **11 critical to medium vulnerabilities**, all remediated following OWASP and CWE best practices. The fixed code provides a solid foundation for secure application development.

Summary of Fixes:

- **5 Critical vulnerabilities** resolved (complete credential compromise, data breach risks eliminated)
- **4 High vulnerabilities** resolved (injection attacks, network security, access control)
- **2 Medium vulnerabilities** resolved (error handling, configuration management)

Key Security Improvements:

1. **Zero hardcoded secrets** - All credentials externalized
2. **Defense in depth** - Multiple layers of security controls
3. **Secure by default** - SSL/TLS validation, HTTPS-only, parameterized queries
4. **Compliance ready** - PCI-DSS, GDPR, SOC2 alignment
5. **Production hardened** - Rate limiting, monitoring, audit logging

Recommendation: Deploy fixed code to staging environment, conduct thorough security validation (penetration testing, code review), then proceed with production deployment following the checklist above.

Next Steps:

1. Review deployment checklist
2. Set up secrets management infrastructure
3. Configure monitoring and alerting
4. Conduct security testing
5. Train development team on secure coding practices
6. Establish ongoing security review process