# Security Test: Vulnerability Analysis and Remediation Report

Security Assessment Submission

November 2025

# Executive Summary

This report presents a comprehensive security analysis of the `Security_Issue_Python_code_unmarked.py` file, which implements a data processing and cloud upload service. The analysis identified **11 critical security vulnerabilities** across multiple categories including secrets management, injection attacks, encryption, input validation, and logging practices.

All identified vulnerabilities have been remediated in the accompanying fixed code. The vulnerabilities range from **Critical** to **Medium** severity, with the most severe issues being hardcoded credentials, SQL injection, and disabled SSL/TLS verification.

**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 | Root Cause | CWE/OWASP Reference | Location |
|---|---|---|---|---|---|
| V-01 | Hardcoded Secrets in Source Code | Critical | API keys, passwords, and AWS credentials stored as plaintext constants | CWE-798: Use of Hard-coded Credentials<br>OWASP A07:2021 – Identification and Authentication Failures | Lines 14-18 |
| V-02 | Secrets Exposed in Application Logs | Critical | Sensitive credentials logged to debug/error logs | CWE-532: Insertion of Sensitive Information into Log File<br>OWASP A09:2021 – Security Logging Failures | Lines 31-32, 62, 131, 160 |
| V-03 | SSL/TLS Certificate Validation Disabled | Critical | All HTTPS requests bypass certificate validation | CWE-295: Improper Certificate Validation<br>OWASP A02:2021 – Cryptographic Failures | Lines 36, 40, 96, 177 |
| V-04 | SQL Injection Vulnerability | Critical | Unsanitized user input concatenated into SQL queries | CWE-89: SQL Injection<br>OWASP A03:2021 – Injection | Lines 71, 172 |
| V-05 | Sensitive Data Stored in Plaintext | Critical | Passwords, credit cards, SSN stored without encryption | CWE-256: Plaintext Storage of Password<br>CWE-359: Exposure of Private Information<br>OWASP A02:2021 – Cryptographic Failures | Lines 49-58 |
| V-06 | Missing Input Validation | High | No validation or sanitization of user inputs and | CWE-20: Improper Input Validation<br>OWASP A03:2021 – Injection | Lines 65-71, 163-183 |

| ID | Issue | Severity | Root Cause | CWE/OWASP Reference | Location |
|---|---|---|---|---|---|
| | | | webhook data | | |
| V-07 | Insecure HTTP Communication | High | Webhook endpoint uses HTTP instead of HTTPS | CWE-319: Cleartext Transmission of Sensitive Information<br>OWASP A02:2021 – Cryptographic Failures | Line 25, 177 |
| V-08 | Missing Rate Limiting and Resource Controls | High | API calls lack rate limiting, enabling DoS attacks | CWE-770: Allocation of Resources Without Limits<br>OWASP A04:2021 – Insecure Design | Lines 83-107 |
| V-09 | Overly Broad Exception Handling | Medium | Generic exception catching hides security issues | CWE-396: Declaration of Catch for Generic Exception<br>OWASP A09:2021 – Security Logging Failures | Lines 61-63, 79-81, 105-107, 130-132, 159-161, 181-183 |
| V-10 | Missing Database Access Controls | High | Database table created without proper permissions or encryption | CWE-732: Incorrect Permission Assignment<br>OWASP A01:2021 – Broken Access Control | Lines 49-58 |
| V-11 | Hardcoded Configuration Values | Medium | Region, URLs, and server addresses hardcoded in source | CWE-547: Use of Hard-coded, Security-relevant Constants<br>OWASP A05:2021 – Security Misconfiguration | Lines 21, 24-25, 117, 139-140 |

---

# 2. Detailed Vulnerability Analysis

## V-01: Hardcoded Secrets in Source Code

### Description

Multiple sensitive credentials are hardcoded as plaintext constants in the source code (lines 14-18): - API key - Database password - AWS access key and secret key - SMTP password

### Impact

**Severity: Critical**

If this code is committed to version control (Git), all credentials become permanently exposed in the repository history. Attackers who gain access to the codebase (via leaked repo, insider threat, or compromised developer machine) can: - Access production databases and exfiltrate sensitive user data - Use AWS credentials to provision resources, leading to financial damage - Send unauthorized emails via SMTP - Make API calls impersonating the service

**Real-world consequence**: Complete compromise of all integrated services and data breaches.

## Evidence

```
# Lines 14-18
API_KEY = "sk-1234567890abcdef1234567890abcdef"
DATABASE_PASSWORD = "admin123"
AWS_ACCESS_KEY = "AKIAIOSFODNN7EXAMPLE"
AWS_SECRET_KEY = "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
SMTP_PASSWORD = "email_password_123"
```

## Fix

**Use environment variables and secrets management systems:** 1. Remove all hardcoded credentials from source code 2. Load secrets from environment variables 3. Use AWS Secrets Manager, HashiCorp Vault, or similar for production 4. Add `.env` to `.gitignore` to prevent accidental commits

## Verification

1. **Code Review**: Search codebase for any hardcoded patterns:

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

2. **Secret Scanning**: Use tools like `git-secrets`, `truffleHog`, or GitHub secret scanning

3. **Runtime Test**: Verify application fails gracefully when environment variables are missing

4. **Credential Rotation**: Immediately rotate all exposed credentials

---

## V-02: Secrets Exposed in Application Logs

### Description

Sensitive credentials are logged to application logs at DEBUG and ERROR levels: - Line 31: API key logged during initialization - Line 32: Database password logged during initialization - Line 62: Database connection string with password logged on error - Line 131: AWS access key logged on S3 upload failure - Line 160: SMTP password logged on email failure

### Impact

**Severity: Critical**

Application logs are typically: - Stored in log aggregation systems (CloudWatch, Splunk, ELK) - Accessible to developers, DevOps, and support teams - Retained for months or years - Sometimes exported to third-party monitoring services

**Consequences**: - Credentials accessible to anyone with log access - Compliance violations (PCI-DSS, HIPAA, GDPR) - Secrets persist even after rotation (in historical logs)

### Evidence

```
        # Lines 31-32
        self.logger.info(f"Initializing with API key: {API_KEY}")
        self.logger.info(f"Database password: {DATABASE_PASSWORD}")

        # Line 62
        self.logger.error(f"Database connection failed: {str(e)} | Connection:
{DB_CONNECTION_STRING}")

        # Line 131
        self.logger.error(f"S3 upload failed: {str(e)} | Credentials:
```

```
{AWS_ACCESS_KEY}")

        # Line 160
        self.logger.error(f"Email failed: {str(e)} | SMTP Password: {SMTP_PASSWORD}")
```

## Fix

1. **Never log secrets**: Remove all credential logging
2. **Log redaction**: Implement automatic redaction for sensitive fields
3. **Structured logging**: Use structured logs with field-level filtering
4. **Sanitized error messages**: Log only non-sensitive error context

## Verification

1. **Log Analysis**: Review all log output for sensitive patterns:

   ```
   python secure_app.py 2>&1 | grep -i "password\|key\|secret"
   ```

2. **Static Analysis**: Use linters with custom rules to detect logging of sensitive variables
3. **Log Inspection**: Check centralized logging systems for exposed credentials
4. **Compliance Scan**: Run PCI-DSS log scanning tools

---

# V-03: SSL/TLS Certificate Validation Disabled

## Description

SSL/TLS certificate validation is disabled throughout the application: - Line 36: `self.session.verify = False` for all requests session - Line 40: urllib3 InsecureRequestWarning disabled - Line 96: `verify=False` in API calls - Line 177: `verify=False` in webhook calls

## Impact

**Severity: Critical**

Disabling certificate validation makes the application vulnerable to **Man-in-the-Middle (MitM) attacks**: - Attackers can intercept HTTPS traffic using self-signed certificates - All data transmitted (including credentials, user data) can be captured - Attackers can modify requests/responses in transit - No guarantee the client is communicating with the legitimate server

**Attack scenario**: Attacker on the same network (coffee shop, compromised router) intercepts API calls and steals API keys or user data.

## Evidence

```
        # Lines 35-40
        self.session = requests.Session()
        self.session.verify = False

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

```
        # Line 96
        response = self.session.post(..., verify=False)

        # Line 177
        response = requests.post(WEBHOOK_ENDPOINT, json=webhook_data, verify=False)
```

## Fix

1. **Enable certificate validation**: Set `verify=True` (default)
2. **Use trusted CA certificates**: Rely on system CA bundle
3. **Custom CA certificates**: If needed, specify certificate path:

   ```
   response = session.post(url, verify='/path/to/ca-bundle.crt')
   ```

4. **Remove warning suppression**: Let urllib3 warnings surface issues

## Verification

1. **Network Interception Test**: Use tools like `mitmproxy` or `Burp Suite` to intercept HTTPS traffic
   - **Expected**: Connection fails with SSL certificate verification error
   - **Vulnerable**: Connection succeeds, traffic is decrypted
2. **Code Audit**: Search for all instances of `verify=False`:

   ```
   grep -r "verify\s*=\s*False" .
   ```

3. **Integration Test**: Ensure app works with valid certificates in production

---

## V-04: SQL Injection Vulnerability

### Description

User input is directly concatenated into SQL queries without parameterization: - Line 71: `user_id` concatenated into SELECT query - Line 172: `user_id` concatenated into DELETE query

This allows attackers to inject arbitrary SQL commands.

### Impact

**Severity: Critical**

SQL injection is one of the most dangerous vulnerabilities (OWASP #3). Attackers can: - **Extract all data**: Dump entire database (user passwords, credit cards, SSNs) - **Modify data**: Update or delete records - **Bypass authentication**: Login as any user - **Execute system commands**: Depending on database permissions - **Denial of Service**: Drop tables or crash the database

**Example attack**:

```
        user_id = "1 OR 1=1"  # Returns all users
        user_id = "1; DROP TABLE user_data; --"  # Deletes table
```

### Evidence

```
        # Line 71
        query = f"SELECT * FROM user_data WHERE id = {user_id}"
        cursor.execute(query)

        # Line 172
```

```
        query = f"DELETE FROM user_data WHERE id = {user_id}"
        cursor.execute(query)
```

## Fix

**Use parameterized queries (prepared statements)**:

```
        # Secure version
        query = "SELECT * FROM user_data WHERE id = ?"
        cursor.execute(query, (user_id,))

        # For DELETE
        query = "DELETE FROM user_data WHERE id = ?"
        cursor.execute(query, (user_id,))
```

**Additional measures**: - Input validation (ensure `user_id` is an integer) - Use ORM (SQLAlchemy) for automatic parameterization - Principle of least privilege (read-only DB user for queries)

## Verification

1. **Penetration Test**: Attempt SQL injection payloads:

```
        processor.fetch_user_data("1 OR 1=1")  # Should fail or return only id=1
        processor.fetch_user_data("1; DROP TABLE user_data; --")  # Should fail
```

2. **Expected Result**: Queries treat entire payload as a single value, not executable SQL
3. **Automated Scanning**: Use SQLMap or OWASP ZAP to test for SQL injection
4. **Code Review**: Ensure all database queries use parameterization

---

## V-05: Sensitive Data Stored in Plaintext

### Description

The database schema (lines 49-58) stores highly sensitive information in plaintext: - User passwords (should be hashed) - Credit card numbers (violates PCI-DSS) - Social Security Numbers (violates privacy regulations)

### Impact

**Severity: Critical**

Storing sensitive data unencrypted leads to: - **Data Breach**: If database is compromised, all sensitive data is exposed - **Compliance Violations**: - PCI-DSS: Credit card data must be encrypted - GDPR/CCPA: PII must be protected - HIPAA: PHI must be encrypted - **Legal Liability**: Fines, lawsuits, reputational damage - **Identity Theft**: SSN and credit card theft affects users

**Industry standards**: - Passwords: Must be hashed with bcrypt/Argon2 - Credit cards: Tokenize via payment gateway or encrypt at rest - SSN: Encrypt at rest with strong encryption (AES-256)

### Evidence

```
# Lines 49-58
cursor.execute("""
    CREATE TABLE IF NOT EXISTS user_data (
        id INTEGER PRIMARY KEY,
        username TEXT,
        password TEXT,          -- PLAINTEXT PASSWORD
        credit_card TEXT,       -- PLAINTEXT CREDIT CARD
        ssn TEXT,               -- PLAINTEXT SSN
```

```
            created_at TIMESTAMP
        )
    """)
```

## Fix

### 1. Password Hashing:

```
    import bcrypt

    # Store hashed password
    hashed_pw = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())
    cursor.execute("INSERT INTO user_data (password) VALUES (?)", (hashed_pw,))

    # Verify password
    if bcrypt.checkpw(input_password.encode('utf-8'), stored_hash):
        # Login successful
```

**2. Credit Card Tokenization**: - Use Stripe, PayPal, or payment gateway tokens - Never store raw credit card numbers - If storage required, use field-level encryption with AWS KMS

### 3. SSN Encryption:

```
    from cryptography.fernet import Fernet

    # Encrypt SSN
    key = os.getenv('ENCRYPTION_KEY')  # Store in AWS KMS
    f = Fernet(key)
    encrypted_ssn = f.encrypt(ssn.encode())

    # Decrypt only when needed
    decrypted_ssn = f.decrypt(encrypted_ssn).decode()
```

**4. Database Encryption**: - Enable encryption at rest (AWS RDS encryption, SQLite encryption extension) - Use database-level field encryption (PostgreSQL pgcrypto)

### Verification

1. **Database Inspection**: Query database and verify:
   - Passwords are bcrypt/Argon2 hashes (unreadable)
   - Credit cards are tokens or encrypted
   - SSN are encrypted blobs
2. **Compliance Scan**: Use PCI-DSS scanning tools
3. **Penetration Test**: Attempt to extract plaintext sensitive data
4. **Code Review**: Ensure all sensitive fields use encryption/hashing before storage

---

## V-06: Missing Input Validation

### Description

No validation or sanitization is performed on user inputs: - Line 65: `user_id` parameter in `fetch_user_data()` not validated - Line 167-172: `webhook_data` fields (`user_id`, `action`) not validated

### Impact

**Severity: High**

Missing input validation enables multiple attack vectors: - **SQL Injection** (as covered in V-04) - **Type Confusion**: Passing objects instead of expected types causes crashes - **Logic Bypass**: Malicious input bypasses business logic - **Command Injection**: If inputs are used in system commands - **DoS**: Malformed input causes exceptions and crashes

### **Example attacks**:

```
# Type confusion
fetch_user_data({"malicious": "dict"})

# Webhook injection
webhook_data = {"user_id": "1 OR 1=1", "action": "../../../etc/passwd"}
```

## Evidence

```
# Lines 65-71 - No validation before SQL query
def fetch_user_data(self, user_id):
    query = f"SELECT * FROM user_data WHERE id = {user_id}"  # Direct use

# Lines 167-172 - No validation on webhook data
user_id = webhook_data.get('user_id')  # Could be None, string, object
action = webhook_data.get('action')
query = f"DELETE FROM user_data WHERE id = {user_id}"  # Unsanitized
```

## Fix

### **Implement strict input validation**:

```
def fetch_user_data(self, user_id):
    # Validate type and range
    if not isinstance(user_id, int):
        try:
            user_id = int(user_id)
        except (ValueError, TypeError):
            raise ValueError("user_id must be an integer")

    if user_id < 1:
        raise ValueError("user_id must be positive")

    # Use parameterized query (defense in depth)
    query = "SELECT * FROM user_data WHERE id = ?"
    cursor.execute(query, (user_id,))

def process_webhook_data(self, webhook_data):
```

```
        # Validate schema
        if not isinstance(webhook_data, dict):
            raise ValueError("webhook_data must be a dictionary")

        # Whitelist allowed actions
        allowed_actions = ['delete_user', 'update_user', 'create_user']
        action = webhook_data.get('action')

        if action not in allowed_actions:
            raise ValueError(f"Invalid action: {action}")

        # Validate user_id
        user_id = webhook_data.get('user_id')
        if not isinstance(user_id, int) or user_id < 1:
            raise ValueError("Invalid user_id")
```

**Additional measures**: - Use validation libraries (Pydantic, Marshmallow) - Define strict schemas for all inputs - Reject unexpected fields (fail closed)

## Verification

1. **Fuzzing**: Send malformed inputs and ensure proper error handling:

   ```
   fetch_user_data(None)  # Should raise error
   fetch_user_data("abc")  # Should raise error
   fetch_user_data(-1)  # Should raise error
   process_webhook_data({"action": "malicious"})  # Should reject
   ```

2. **Unit Tests**: Test boundary conditions (negative, zero, large numbers, None, objects)

3. **Integration Tests**: Ensure API rejects invalid payloads with 400 Bad Request

4. **Security Scan**: Use OWASP ZAP or Burp Suite to test input vectors

---

# V-07: Insecure HTTP Communication

## Description

The webhook endpoint uses HTTP instead of HTTPS (line 25):

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

## Impact

**Severity: High**

HTTP transmits data in **cleartext**, allowing attackers to: - **Eavesdrop**: Capture all webhook data in transit - **Man-in-the-Middle**: Modify webhook payloads before delivery - **Data Leakage**: Sensitive user data (user_id, action) exposed on network - **Credential Theft**: If webhook includes authentication tokens

Even on “internal” networks, attacks are possible via: - Compromised internal machines - Malicious insiders - Network misconfiguration

## Evidence

```
        # Line 25
        WEBHOOK_ENDPOINT = "http://internal-webhook.company.com/process"

        # Line 177
        response = requests.post(WEBHOOK_ENDPOINT, json=webhook_data, verify=False)
```

## Fix

1. **Use HTTPS**: Change URL to `https://internal-webhook.company.com/process`
2. **Enable certificate validation**: Remove `verify=False`
3. **Mutual TLS**: For internal services, use mTLS for authentication
4. **VPN/Private Network**: Ensure internal services are not internet-accessible

```
        WEBHOOK_ENDPOINT = "https://internal-webhook.company.com/process"
        response = requests.post(WEBHOOK_ENDPOINT, json=webhook_data, timeout=10)
```

## Verification

1. **Network Capture**: Use Wireshark to capture traffic:
   - **Vulnerable (HTTP)**: Payload visible in plaintext
   - **Secure (HTTPS)**: Payload encrypted (TLS handshake visible)
2. **Endpoint Test**: Ensure webhook endpoint only accepts HTTPS connections
3. **Certificate Validation**: Test that invalid certificates are rejected

---

# V-08: Missing Rate Limiting and Resource Controls

## Description

The `call_external_api()` method (lines 83-107) has no rate limiting, retry logic, or resource controls. Comment on line 84 acknowledges this: “Make API calls without proper error handling or rate limiting.”

## Impact

**Severity: High**

Lack of rate limiting enables: - **Denial of Service**: Attackers can flood API with requests, exhausting resources - **Cost Explosion**: Cloud API calls are metered; unlimited calls = unlimited costs - **Service Degradation**: Excessive requests slow down the application - **IP Blocking**: External APIs may block the service for abuse - **Resource Exhaustion**: Memory/connection pool exhaustion

**Example**: A malicious user triggers 10,000 API calls in 1 minute, costing thousands in API fees.

## Evidence

```
        # Lines 83-107
        def call_external_api(self, data):
            """Make API calls without proper error handling or rate limiting"""
            # No rate limiting
            # No retry logic
            # No timeout
            response = self.session.post(...)
```

## Fix

### Implement rate limiting and resource controls:

```
        from time import sleep, time
        from collections import deque
        import requests

        class DataProcessor:
            def __init__(self):
                self.api_calls = deque()
                self.max_calls_per_minute = 60
                self.timeout = 10  # seconds

            def call_external_api(self, data):
                # Rate limiting (token bucket algorithm)
                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()

                # Check rate limit
                if len(self.api_calls) > self.max_calls_per_minute:
                    sleep(1)  # Back off
                    raise Exception("Rate limit exceeded")

                # Retry logic with exponential backoff
                for attempt in range(3):
                    try:
                        response = self.session.post(
                            f"{API_BASE_URL}/process",
                            headers=headers,
                            json=data,
                            timeout=self.timeout
                        )
                        response.raise_for_status()
                        return response.json()
                    except requests.exceptions.Timeout:
                        if attempt < 2:
                            sleep(2 ** attempt)  # 1s, 2s, 4s
                            continue
```

```
                raise
            except requests.exceptions.HTTPError as e:
                if e.response.status_code == 429:  # Too Many Requests
                    sleep(60)
                    continue
                raise
```

**Use production-grade rate limiting**: - **Redis-based rate limiting**: For distributed systems - **API Gateway**: AWS API Gateway has built-in rate limiting - **Third-party libraries**: `ratelimit`, `python-ratelimit`

### Verification

1. **Load Test**: Use Apache JMeter or Locust to send 1000 requests/second
   - **Expected**: Requests are rate-limited (429 status or backoff)
2. **Stress Test**: Monitor memory and connection pool usage under load
3. **Cost Monitoring**: Set AWS billing alarms for unexpected API usage
4. **Unit Test**: Verify rate limiter blocks excessive calls

---

## V-09: Overly Broad Exception Handling

### Description

The application catches generic `Exception` throughout the codebase without proper handling: - Lines 61-63: Database connection errors - Lines 79-81: Query execution errors - Lines

105-107: API request errors - Lines 130-132: S3 upload errors - Lines 159-161: Email sending errors - Lines 181-183: Webhook processing errors

## Impact

**Severity: Medium**

Broad exception handling creates several problems: - **Masked Bugs**: Hides programming errors and security issues - **Poor Observability**: Generic error messages don't identify root cause - **Silent Failures**: Operations fail without proper alerts - **Security Bypass**: Authentication/authorization errors silently caught

**Example**: A KeyError (wrong variable name) is caught as generic Exception, making debugging impossible.

## Evidence

```
        # Lines 61-63
        try:
            # ... database code
        except Exception as e:  # Too broad
            self.logger.error(f"Database connection failed: {str(e)}")
            return None, None
```

## Fix

**Use specific exception handling**:

```
        import sqlite3

        def connect_to_database(self):
            try:
                conn = sqlite3.connect("app_data.db", timeout=10)
```

```
            cursor = conn.cursor()
            # ... table creation
            return conn, cursor
        except sqlite3.Error as e:
            # Specific database errors
            self.logger.error(f"Database error: {type(e).__name__}: {e}")
            raise
        except PermissionError as e:
            # File permission errors
            self.logger.error(f"Permission denied accessing database: {e}")
            raise
        except Exception as e:
            # Unexpected errors - should be investigated
            self.logger.critical(f"Unexpected error in connect_to_database: {e}")
            raise

    def call_external_api(self, data):
        try:
            response = self.session.post(...)
            response.raise_for_status()
            return response.json()
        except requests.exceptions.Timeout as e:
            self.logger.error(f"API timeout: {e}")
            raise
        except requests.exceptions.ConnectionError as e:
            self.logger.error(f"API connection failed: {e}")
            raise
        except requests.exceptions.HTTPError as e:
            self.logger.error(f"API HTTP error {e.response.status_code}: {e}")
            raise
        except requests.exceptions.RequestException as e:
            self.logger.error(f"API request failed: {e}")
            raise
```

**Best practices**: - Catch specific exceptions only - Re-raise unexpected exceptions - Use finally blocks for cleanup - Log exception types and context

### Verification

1. **Code Review**: Search for generic `except Exception` or bare `except:`
2. **Exception Testing**: Trigger each exception type and verify proper handling
3. **Monitoring**: Configure alerts for critical/unexpected exceptions
4. **Log Analysis**: Review logs to ensure exception types are identifiable

---

## V-10: Missing Database Access Controls

### Description

The database table is created without: - Row-level security - Column-level permissions - Encryption at rest - Audit logging

Comment on line 48 acknowledges: “Creating table without proper permissions/security”

### Impact

**Severity: High**

Missing access controls allow: - **Privilege Escalation**: Any database user can access all data - **Data Leakage**: No enforcement of least privilege principle - **Compliance Violations**: Regulations require access controls on sensitive data - **No Audit Trail**: Can’t track who accessed what data

## Evidence

```
        # Lines 48-58
        # Creating table without proper permissions/security
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS user_data (
                ...
                password TEXT,      -- No encryption
                credit_card TEXT,   -- No encryption or access control
                ssn TEXT            -- No encryption or access control
            )
        """)
```

## Fix

### Implement database security controls:

### 1. Use Database Roles and Permissions:

```
        -- PostgreSQL example
        CREATE ROLE app_read WITH LOGIN PASSWORD 'secure_password';
        CREATE ROLE app_write WITH LOGIN PASSWORD 'secure_password';

        -- Grant minimal permissions
        GRANT SELECT ON user_data TO app_read;
        GRANT SELECT, INSERT, UPDATE ON user_data TO app_write;

        -- Row-level security
        ALTER TABLE user_data ENABLE ROW LEVEL SECURITY;

        CREATE POLICY user_isolation ON user_data
            FOR SELECT
            USING (user_id = current_user_id());
```

### 2. Encrypt Sensitive Columns:

```
        -- PostgreSQL with pgcrypto
        CREATE EXTENSION IF NOT EXISTS pgcrypto;
```

```
CREATE TABLE user_data (
    id SERIAL PRIMARY KEY,
    username TEXT NOT NULL,
    password_hash TEXT NOT NULL,  -- bcrypt hash
    credit_card_encrypted BYTEA,  -- Encrypted with pgp_sym_encrypt
    ssn_encrypted BYTEA,          -- Encrypted
    created_at TIMESTAMP DEFAULT NOW()
);

-- Encrypt on insert
INSERT INTO user_data (credit_card_encrypted)
VALUES (pgp_sym_encrypt('4111111111111111', 'encryption_key'));

-- Decrypt on read (with proper authorization)
SELECT pgp_sym_decrypt(credit_card_encrypted, 'encryption_key') FROM user_data;
```

**3. Enable Audit Logging**:

```
-- Enable PostgreSQL audit logging
ALTER SYSTEM SET log_statement = 'mod';  -- Log all modifications
ALTER SYSTEM SET log_connections = 'on';
```

**4. Application-Level Controls**: - Use separate database users for read vs. write operations - Implement application-level access control lists (ACLs) - Use ORMs with built-in permission systems

## Verification

1. **Permission Test**: Attempt unauthorized access with restricted user:

   ```
   -- As app_read user
   DELETE FROM user_data WHERE id = 1;  -- Should fail
   ```

2. **Encryption Verification**: Query database directly and verify encrypted fields are unreadable

3. **Audit Log Review**: Check database logs for all access events
4. **Compliance Scan**: Run database security scanners (AWS Inspector, pgAudit)

---

## V-11: Hardcoded Configuration Values

### Description

Multiple configuration values are hardcoded in the source code: - Line 21: Database connection string with hostname - Line 24: API base URL - Line 25: Webhook endpoint - Line 117: AWS region - Lines 139-140: SMTP server and port

### Impact

**Severity: Medium**

Hardcoded configuration creates operational challenges: - **Environment Parity**: Same code can't run in dev/staging/production - **Deployment Complexity**: Code changes required for configuration updates - **Security Risk**: Configuration embedded in version control - **Vendor Lock-in**: Difficult to switch services/regions

### Evidence

```
        # Line 21
        DB_CONNECTION_STRING = f"postgresql://admin:{DATABASE_PASSWORD}@prod-db.company.com:5432/maindb"

        # Lines 24-25
        API_BASE_URL = "https://api.production-service.com/v1"
```

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

        # Line 117
        region_name='us-east-1'  # Hardcoded region

        # Lines 139-140
        smtp_server = "smtp.gmail.com"
        smtp_port = 587
```

## Fix

**Use environment-based configuration**:

```
        import os
        from dotenv import load_dotenv

        load_dotenv()  # Load from .env file

        # Configuration from environment
        DB_HOST = os.getenv('DB_HOST', 'localhost')
        DB_PORT = os.getenv('DB_PORT', '5432')
        DB_NAME = os.getenv('DB_NAME', 'maindb')
        DB_USER = os.getenv('DB_USER')
        DB_PASSWORD = os.getenv('DB_PASSWORD')

        DB_CONNECTION_STRING = f"postgresql://{DB_USER}:{DB_PASSWORD}@{DB_HOST}:
{DB_PORT}/{DB_NAME}"

        API_BASE_URL = os.getenv('API_BASE_URL')
        WEBHOOK_ENDPOINT = os.getenv('WEBHOOK_ENDPOINT')

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

        SMTP_SERVER = os.getenv('SMTP_SERVER', 'smtp.gmail.com')
        SMTP_PORT = int(os.getenv('SMTP_PORT', '587'))
```

**Production configuration management**: - AWS Systems Manager Parameter Store - AWS Secrets Manager (for secrets) - Kubernetes ConfigMaps and Secrets - HashiCorp Consul

**Example .env file** (never commit to Git):

```
DB_HOST=prod-db.company.com
DB_USER=app_user
DB_PASSWORD=<stored-in-secrets-manager>
API_BASE_URL=https://api.production-service.com/v1
AWS_REGION=us-east-1
```

## Verification

1. **Multi-Environment Test**: Deploy to dev, staging, and production with different configs
2. **Configuration Validation**: Ensure app fails early if required configs are missing
3. **Git History**: Verify no `.env` files committed to version control
4. **Deployment Test**: Change configuration without code changes

---

# 3. Summary of Remediations

## Code Changes Overview

The fixed code (`security_fixed_code.py`) implements the following security improvements:

### Secrets Management

- ✅ Removed all hardcoded credentials
- ✅ Loaded secrets from environment variables
- ✅ Added support for AWS Secrets Manager
- ✅ Removed all credential logging

### Encryption & Communication

- ✅ Enabled SSL/TLS certificate validation (removed `verify=False`)
- ✅ Changed webhook endpoint to HTTPS
- ✅ Implemented password hashing with bcrypt
- ✅ Added field-level encryption for sensitive data (SSN, credit cards)

### Input Validation & Injection Prevention

- ✅ Implemented parameterized SQL queries
- ✅ Added input validation with type checking and whitelisting
- ✅ Used Pydantic models for webhook schema validation

## Error Handling & Logging

- ✅ Replaced generic exceptions with specific exception types
- ✅ Removed sensitive data from all log messages
- ✅ Implemented structured logging with field redaction

## Resource Controls

- ✅ Added rate limiting with token bucket algorithm
- ✅ Implemented retry logic with exponential backoff
- ✅ Added request timeouts
- ✅ Implemented connection pooling limits

## Database Security

- ✅ Implemented database encryption at rest
- ✅ Added role-based access control recommendations
- ✅ Enabled audit logging
- ✅ Used least privilege database users

## Configuration Management

- ✅ Externalized all configuration to environment variables
- ✅ Added configuration validation
- ✅ Implemented multi-environment support

## Security Testing Recommendations

### Automated Security Scanning

1. **Static Analysis**: Bandit, Semgrep, SonarQube
2. **Dependency Scanning**: Safety, pip-audit, Snyk
3. **Secret Scanning**: git-secrets, TruffleHog, GitHub Advanced Security
4. **Container Scanning**: Trivy, Clair (if containerized)

### Penetration Testing

1. **SQL Injection**: SQLMap, manual testing
2. **API Security**: OWASP ZAP, Burp Suite
3. **Authentication**: Test credential rotation, session management
4. **Network Security**: Wireshark, mitmproxy for MitM testing

### Compliance Validation

1. **PCI-DSS**: Credit card data handling
2. **GDPR**: Personal data encryption and access controls
3. **SOC 2**: Audit logging and access control
4. **HIPAA**: If handling health information

## Deployment Checklist

Before deploying the fixed code to production:

☐

- [ ] Rotate all exposed credentials (API keys, passwords, AWS keys)
- [ ] Configure AWS Secrets Manager with production secrets
- [ ] Set up environment-specific configuration files
- [ ] Enable database encryption at rest
- [ ] Configure rate limiting thresholds
- [ ] Set up centralized logging with field redaction
- [ ] Enable AWS GuardDuty and Security Hub
- [ ] Conduct penetration testing
- [ ] Review and approve database permissions
- [ ] Configure monitoring and alerting
- [ ] Train team on secure coding practices
- [ ] Document incident response procedures

---

# 4. Conclusion

This security assessment identified **11 critical to medium vulnerabilities** in the original code, primarily related to secrets management, injection attacks, and encryption practices. All vulnerabilities have been remediated following industry best practices and security standards (OWASP, CWE, NIST).

The fixed code provides a solid foundation for secure application development, but security is an ongoing process requiring: - Regular security audits and penetration testing - Continuous monitoring and logging - Timely patching of dependencies - Security awareness training for development teams - Incident response planning

**Recommendation**: Deploy the fixed code to a staging environment, conduct thorough integration testing and security validation before production deployment.