Security Assessment Report

1. Executive Summary

This assessment identified multiple critical security vulnerabilities in the `DataProcessor` application, including hardcoded credentials, SQL injection flaws, and insecure network communications. Immediate remediation is required to prevent data breaches and unauthorized access.

2. Vulnerability Analysis

2.1 Hardcoded Credentials (Critical)

- **Location:** Lines 14-18, 21, 139-141.
- **Issue:** API keys, database passwords, AWS secrets, and SMTP credentials are stored in plain text within the source code.
- **Risk:** Anyone with access to the code (including via version control history) can compromise the production database, cloud storage, and email service.
- **Remediation:** Use Environment Variables or a Secrets Manager (e.g., AWS Secrets Manager, HashiCorp Vault).

2.2 SQL Injection (Critical)

- **Location:** Lines 71 (`fetch_user_data`) and 172 (`process_webhook_data`).
- **Issue:** User input (`user_id`) is directly concatenated into SQL queries using f-strings.
- **Risk:** An attacker can manipulate the `user_id` to execute arbitrary SQL commands, potentially dumping the entire database or deleting data.
- **Remediation:** Use parameterized queries (prepared statements) provided by the `sqlite3` library (e.g., `cursor.execute("SELECT * FROM user_data WHERE id = ?", (user_id,))`).

2.3 Insecure SSL/TLS Verification (High)

- **Location:** Lines 36, 40, 96, 177.
- **Issue:** `verify=False` is used in `requests` calls, and warnings are suppressed.
- **Risk:** This disables certificate validation, making the application vulnerable to Man-in-the-Middle (MitM) attacks. An attacker can intercept and modify traffic between the app and the API.

- **Remediation:** Remove `verify=False` and ensure the host system has the correct CA certificates installed.

2.4 Sensitive Data Logging (High)

- **Location:** Lines 31, 32, 62, 131, 160.
- **Issue:** The application logs sensitive information, including the API Key, Database Password, and Connection String (which contains the password).
- **Risk:** Logs are often stored in less secure locations (e.g., plain text files, centralized logging systems). An attacker gaining access to logs would obtain valid credentials.
- **Remediation:** Remove logging statements that output secrets. Use a redaction filter if necessary.

2.5 Insecure Data Storage (High)

- **Location:** Lines 50-57.
- **Issue:** The `user_data` table stores passwords, credit card numbers, and SSNs in plain text (`TEXT` fields).
- **Risk:** If the database is compromised (e.g., via the SQL injection above), all user PII and credentials are instantly readable.
- **Remediation:** Hash passwords using a strong algorithm (Argon2/bcrypt). Encrypt sensitive fields (SSN, Credit Card) at rest.

2.6 Lack of Input Validation (Medium)

- **Location:** `process_webhook_data` (Line 163).
- **Issue:** The method accepts `webhook_data` and blindly trusts the `action` and `user_id` without validating their type or format.
- **Risk:** Can lead to logic errors or exploitation of the SQL injection vulnerability.
- **Remediation:** Validate that `user_id` is an integer and `action` is a valid enum string before processing.

3. Remediation Code (Fixed)

```
import os
import logging
import sqlite3
import requests
import boto3
import smtplib
from email.mime.text import MIMEText
```

```
# Load configuration from Environment Variables
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")
SMTP_PASSWORD = os.getenv("SMTP_PASSWORD")
DB_CONNECTION_STRING = os.getenv("DB_CONNECTION_STRING")

class SecureDataProcessor:
    def __init__(self):
        logging.basicConfig(level=logging.INFO)
        self.logger = logging.getLogger(__name__)

        if not all([API_KEY, DATABASE_PASSWORD, AWS_ACCESS_KEY]):
            self.logger.critical("Missing required environment variables!")
            raise EnvironmentError("Security credentials not found.")

        self.session = requests.Session()
        # REMOVED: verify=False and warning suppression

    def connect_to_database(self):
        try:
            conn = sqlite3.connect("app_data.db")
            cursor = conn.cursor()
            # Fixed: Passwords/SSN should be encrypted/hashed (Schema update required in real scenario)
            return conn, cursor
        except Exception as e:
            # Fixed: Do not log the connection string containing passwords
            self.logger.error(f"Database connection failed: {str(e)}")
            return None, None

    def fetch_user_data(self, user_id):
        conn, cursor = self.connect_to_database()
        if not cursor:
            return None

        # Fixed: SQL Injection (Parameterized Query)
        query = "SELECT * FROM user_data WHERE id = ?"

        try:
            # Validate input type
            if not isinstance(user_id, int):
                raise ValueError("Invalid user_id")

            cursor.execute(query, (user_id,))
            result = cursor.fetchone()
            return result
        except Exception as e:
            self.logger.error(f"Query failed: {e}")
            return None
        finally:
            if conn: conn.close()

    def call_external_api(self, data):
        headers = {
            'Authorization': f'Bearer {API_KEY}',
            'Content-Type': 'application/json'
        }

        try:
            # Fixed: SSL Verification enabled by default
            response = self.session.post(
                f"{os.getenv('API_BASE_URL')}/process",
                headers=headers,
                json=data
            )
            response.raise_for_status()
            return response.json()
        except Exception as e:
            self.logger.error(f"API request failed: {str(e)}")
            return None

    def process_webhook_data(self, webhook_data):
        try:
            user_id = webhook_data.get('user_id')
            action = webhook_data.get('action')

            # Input Validation
            if not isinstance(user_id, int):
                return {"status": "error", "message": "Invalid user_id"}

            if action == 'delete_user':
                conn, cursor = self.connect_to_database()
                # Fixed: SQL Injection
                cursor.execute("DELETE FROM user_data WHERE id = ?", (user_id,))
                conn.commit()
                conn.close()

            return {"status": "processed"}

        except Exception as e:
            self.logger.error(f"Webhook processing failed: {str(e)}")
            return {"status": "error"}
```