Security Assessment Report

Target File: `Security_Issue_Python_code_unmarked.py`

Date: December 20, 2025

Assessor: Matthew

---

1. Vulnerability Table

ID	Issue	Severity	Root Cause	
V-01	Hardcoded Secrets	**Critical**	Credentials stored in plaintext global variables.	CV
V-02	SQL Injection	**Critical**	Dynamic SQL construction using f-strings.	CV
V-03	Sensitive Data Exposure in Logs	**High**	Logging credentials and connection strings.	CV
V-04	Disabled SSL Verification	**High**	Disabling SSL cert checks (`verify=False`).	CV
V-05	Plaintext Storage of PII	**High**	Storing SSN/Passwords without hashing/encryption.	CV
V-06	Lack of Input Validation	**High**	Webhook input processed without type checking.	CV
V-07	Broken Access Control (Webhook)	**Critical**	Critical actions (`delete_user`) performed without auth.	CV
V-08	Missing Request Timeouts	**Medium**	Network calls lack `timeout` (DoS risk).	CV
V-09	Debug Mode Enabled	**Medium**	Verbose logging enabled in production scope.	CV

---

2. Detailed Explanations

V-01: Hardcoded Secrets

- **Description:** API keys, database passwords, AWS keys, and SMTP credentials are defined as global variables in the source code.
- **Impact:** If this code is leaked or committed to a repository, attackers can scrape these keys to access production databases, compromise AWS cloud resources, and hijack the SMTP server.
- **Evidence:** `API_KEY = "sk-123..."` (Line 14); `AWS_ACCESS_KEY = "..."` (Line 16).
- **Fix:** Replace hardcoded strings with environment variables using `os.getenv()`. Use a `.env` file for local development and a Secrets Manager for production.
- **Verification:** Run the script with the environment variables unset; the application should raise a configuration error rather than connecting with default hardcoded keys.

V-02: SQL Injection

- **Description:** The code uses Python f-strings to insert user input directly into SQL queries without sanitization.
- **Impact:** An attacker can manipulate input (e.g., `user_id`) to execute arbitrary SQL commands. This could lead to full database dumps or data destruction (e.g., `; DROP TABLE user_data`).
- **Evidence:** `query = f"SELECT * FROM user_data WHERE id = {user_id}"` (Line 68).
- **Fix:** Use **Parameterized Queries** where the database driver handles escaping. Example: `cursor.execute("SELECT * FROM user_data WHERE id = ?", (user_id,))`.
- **Verification:** Attempt to inject a payload like `'1 OR 1=1'` as the `user_id`. The fixed code should treat it as a literal string (and fail/return empty) rather than executing the logic.

V-03: Sensitive Data Exposure in Logs

- **Description:** The application explicitly logs sensitive information, including passwords and connection strings, to the console/log file.
- **Impact:** Logs are often stored in semi-public locations (like centralized logging servers). Anyone with read access to logs can view plaintext admin passwords.
- **Evidence:** `self.logger.info(f"Database password: {DATABASE_PASSWORD}")` (Line 38); `self.logger.error(... {SMTP_PASSWORD})` (Line 158).
- **Fix:** Remove all logging statements that include secret variables. Ensure exception handling logs the error message but scrubs the credentials.
- **Verification:** Force a database connection failure. Check the logs to ensure the password is no longer visible in the error traceback.

V-04: Disabled SSL Verification

- **Description:** The code globally disables SSL verification (`verify=False`) and suppresses security warnings.
- **Impact:** This makes the application vulnerable to **Man-in-the-Middle (MitM)** attacks. An attacker on the network can intercept, read, and modify traffic between the app and external APIs.
- **Evidence:** `self.session.verify = False` (Line 44); `urllib3.disable_warnings` (Line 48).
- **Fix:** Remove `verify=False` and the warning suppression. Ensure the host system has valid CA certificates installed.

- **Verification:** Use a proxy (like Burp Suite) with a self-signed certificate to intercept the request. The fixed application should refuse the connection.

V-05: Plaintext Storage of PII

- **Description:** The database schema defines `password`, `credit_card`, and `ssn` columns as plain `TEXT` without encryption or hashing.
- **Impact:** A database breach results in the immediate exposure of users' financial and personal identity information.
- **Evidence:** `CREATE TABLE... password TEXT, credit_card TEXT, ssn TEXT` (Lines 53–59).
- **Fix:** Store passwords using a strong hash (e.g., bcrypt). Encrypt PII (SSN, Credit Card) at the application level before insertion using a library like `cryptography`.
- **Verification:** Inspect the generated `app_data.db` file. The sensitive columns should contain unreadable hashes/ciphertext, not plaintext data.

V-06: Lack of Input Validation

- **Description:** The `process_webhook_data` function processes external input (`user_id`) without verifying that it is the correct type (integer).
- **Impact:** Passing objects or strings into functions expecting integers can cause logic crashes or unexpected behavior in the database layer.
- **Evidence:** `user_id = webhook_data.get('user_id')` (Line 139) is used immediately without validation.
- **Fix:** Implement strict type checking: `if not isinstance(user_id, int): raise ValueError(...)`.
- **Verification:** Send a payload with `"user_id": "malicious_string"`. The fixed code should catch the error and reject the request safely.

V-07: Broken Access Control (Webhook)

- **Description:** The webhook endpoint allows the destructive `delete_user` action based solely on the JSON payload, without verifying the sender's identity.
- **Impact:** Any attacker who can reach the endpoint can delete any user account by sending a simple JSON request.
- **Evidence:** Lines 132-140 process the action immediately without checking for an API token or HMAC signature.
- **Fix:** Require a shared secret (e.g., `X-Webhook-Token` header) and verify it matches the server's expected secret before processing actions.

- **Verification:** Send a request without the auth token. The system should return a 401/403 Forbidden error.

V-08: Missing Request Timeouts

- **Description:** The application makes synchronous network calls (`requests.post`) without defining a `timeout`.
- **Impact:** If the external service hangs or drops packets, the application thread will hang indefinitely, potentially leading to Resource Exhaustion (DoS).
- **Evidence:** `self.session.post(..., verify=False)` (Line 83) lacks a timeout argument.
- **Fix:** Add `timeout=10` (or appropriate limit) to all network calls.
- **Verification:** Simulate a network delay/drop. The application should raise a `Timeout` exception after the specified limit rather than hanging forever.

V-09: Debug Mode Enabled

- **Description:** The logging configuration is set to `logging.DEBUG` level.
- **Impact:** Debug logs are too verbose for production and may inadvertently capture session headers, variable states, or other sensitive transient data.
- **Evidence:** `logging.basicConfig(level=logging.DEBUG)` (Line 35).
- **Fix:** Set the logging level to `INFO` or `WARNING` for production environments.
- **Verification:** Run the application. Only high-level operational messages should appear in the logs, not granular debug steps.