ENTERPRISE SECURITY AUDIT REPORT

Application Name: Enterprise Order Management System (EOMS)

Developed Using: Python (Flask Framework), SQLite3, AWS SDK

Reviewed By: Intern – shiva kumar

Date of Review: 19 December 2025

Type of Review: Manual & Automated (Static Analysis)

1. Executive Summary

The purpose of this review was to identify critical security weaknesses in the legacy order processing system (`OrderManagement_Legacy.py`) and provide remediation recommendations. The audit focused on preventing Remote Code Execution (RCE), SQL Injection, and Sensitive Data Exposure.

Risk Summary:

- **Total Vulnerabilities:** 5
- **Critical Severity:** 2 (Immediate Action Required)
- **High Severity:** 2
- **Medium Severity:** 1

2. Scope & Methodology

- **Files Scanned:** `OrderManagement_Legacy.py`, `admin_utils.py`.
- **Tools Used:** Bandit, SonarQube, Manual Inspection.
- **Methodology:**
    - **Automated Scan:** Bandit was executed from the project root to identify hardcoded secrets and dangerous function calls.

- **Manual Review:** Line-by-line analysis of authentication logic, database queries, and file handling operations.

3. Vulnerabilities Summary

S.No	Vulnerability	Location/Route	Description	Recommendation
1	**Insecure Deserialization**	`/restore_session`	Usage of `pickle` allows arbitrary code execution if the cookie is modified.	Replace `pickle` with `JSON` or signing mechanisms.
2	**OS Command Injection**	`/system_check`	User input is passed directly to the OS shell via `os.popen`	Use `subprocess` module with input validation.
3	**Path Traversal (LFI)**	`/view_receipt`	Lack of filename validation allows access to system files (e.g., `/etc/passwd`)	Sanitize filenames using `secure_filename`.
4	**SQL Injection**	`/get_order`	Direct string concatenation in SQL queries.	Use Parameterized Queries (Prepared Statements).
5	**Hardcoded Secrets**	Global Config	AWS Access Keys are stored in plain text.	Use Environment Variables or Vault.

4. Detailed Findings & Remediation

4.1. Insecure Deserialization (RCE)

Severity: Critical **Location:** `OrderManagement_Legacy.py` (Line 49)

Description: The application uses the `pickle` library to deserialize session cookies. Since `pickle` allows the execution of arbitrary Python code during deserialization, an attacker can craft a malicious cookie to gain full control over the server.

Vulnerable Code:

```
# VULNERABLE
import pickle
decoded = base64.b64decode(cookie_data)
user_obj = pickle.loads(decoded) # EXECUTES MALICIOUS CODE
```

4.2. OS Command Injection

Severity: Critical **Location:** `/admin/system_check`

Description: The application takes an IP address from the user and concatenates it into a shell command. An attacker can input `8.8.8.8; rm -rf /` to delete server files.

Vulnerable Code:

```
# VULNERABLE
ip = request.form['ip']
os.popen('ping -c 1 ' + ip)
```

4.3. SQL Injection

Severity: High **Location:** `/get_order`

Description: The application constructs SQL queries using f-strings or concatenation. This allows attackers to manipulate the query logic (e.g., dumping the entire database).

Vulnerable Code:

```
# VULNERABLE
query = "SELECT * FROM orders WHERE id = " + order_id
cursor.execute(query)
```

4.4. Hardcoded Secrets

Severity: Medium **Location:** `OrderManagement_Legacy.py` (Line 15)

Description: AWS Access Keys and Secret Keys are hardcoded in the source code. If this code is pushed to a repository, the credentials will be compromised.

Vulnerable Code:

```
# VULNERABLE
AWS_ACCESS_KEY = "AKIA1234567890FAKEKEY"
```

5. Appendix: The Secured Source Code (Remediation)

Below is the complete, fixed version of the application code.

```python

---

# FILE: OrderManagement_Secure.py

# STATUS: PATCHED & SECURE

---

import sqlite3

import json # FIX: Replaced pickle with json

import os

import subprocess

import base64 from flask

import Flask, request, abort from werkzeug.utils
```

import secure_filename # FIX: For Path Traversal

app = Flask(**name**)

FIX: Loaded Secrets from Environment Variables

AWS_ACCESS_KEY = os.environ.get("AWS_ACCESS_KEY") app.config['DEBUG'] = False # FIX: Disabled debug mode

def db_connect(): return sqlite3.connect('orders.db')

@app.route('/get_order', methods=['GET']) def get_order(): order_id = request.args.get('id')

```
# FIX: Input Validation
if not order_id or not order_id.isdigit():
    abort(400, "Invalid Order ID")

try:
    conn = db_connect()
    cursor = conn.cursor()

    # FIX: Parameterized Query (Prevents SQL Injection)
    query = "SELECT * FROM orders WHERE id = ?"
    cursor.execute(query, (order_id,))
    data = cursor.fetchall()
    return str(data)
except Exception:
    return "Database Error", 500
finally:
    conn.close()
```

@app.route('/admin/system_check', methods=['POST']) def system_check(): ip_address = request.form.get('ip')

```
if not ip_address:
    return "IP Required", 400

# FIX: Use subprocess without shell (Prevents Command Injection)
try:
    result = subprocess.run(
```

```
        ['ping', '-c', '1', ip_address],
        capture_output=True,
        text=True,
        timeout=5
    )
    return result.stdout
except Exception:
    return "Ping Failed"
```

@app.route('/restore_session', methods=['POST']) def restore_session(): cookie_data = request.form.get('session_cookie') if not cookie_data: return "No cookie provided", 400

```
try:
    decoded = base64.b64decode(cookie_data)

    # FIX: Use JSON instead of Pickle (Prevents RCE)
    user_obj = json.loads(decoded)
    return f"Welcome back, {user_obj.get('username', 'Guest')}"
except:
    return "Invalid Session"
```

@app.route('/view_receipt', methods=['GET']) def view_receipt(): filename = request.args.get('file') if not filename: abort(400)

```
# FIX: Path Traversal Protection
safe_name = secure_filename(filename)
base_dir = os.path.abspath("receipts")
file_path = os.path.join(base_dir, safe_name)

if not file_path.startswith(base_dir) or not
os.path.exists(file_path):
    abort(403, "Access Denied")

with open(file_path, 'r') as f:
    return f.read()
```

if **name** == '**main**': app.run(host='0.0.0.0', port=5000)

6. Conclusion

The application `OrderManagement_Legacy.py` contained critical vulnerabilities including Remote Code Execution and SQL Injection. These have been remediated in the attached secure code. The system is now compliant with secure coding standards.