

Security Test Report

1. Vulnerability Table

ID	ISSUE	SEVERITY	ROOT CAUSE	CWE/OWASP Reference	LOCATION
1	Hardcoded API key	High	Secret embedded directly in source code	CWE-798 (Use of Hardcoded Credentials)	Top of file
2	Hardcoded Database Password	High	Credentials stored in code	CWE-798	Top of file
3	Hardcoded AWS Access Key & Secret	Critical	Cloud keys exposed in source	CWE-798 / OWASP A2: Sensitive Data Exposure	Top of file
4	Hardcoded SMTP Password	High	Hardcoded email credentials	CWE-798	Top of file
5	Logging sensitive secrets	High	Logs reveal credentials	CWE-532 (Info Exposure in Logs)	<code>__init__()</code>
6	TLS certificate verification disabled	Critical	verify=False allows MITM	CWE-295 (Improper Certificate Validation)	Requests Session
7	SQL Injection	Critical	Unparameterized SQL query	CWE-89 / OWASP A1 Injection	<code>fetch_user_data()</code>
8	Storing plaintext passwords	High	No hashing mechanism	CWE-256 (Plaintext Storage of Passwords)	DB schema
9	Storing	Critical	No encryption	CWE-311 /	DB schema

	credit cards unencrypted		at rest	PCI DSS Violation	
1 0	Storing SSNs unencrypted	Critical	Sensitive PII not protected	CWE-359	DB schema
1 1	Database connection string leaked in logs	High	Overly verbose error logging	CWE-209	connect_to_database()
	No API rate limiting	Medium	Unlimited outbound calls	CWE-770	call_external_api()
	API requests made without TLS verification	High	MITM vulnerability	CWE-295	call_external_api()
	AWS S3 upload uses hardcoded credentials	Critical	Credential exposure	CWE-798	upload_to_cloud()
	Sensitive S3 bucket name exposed	Medium	Overexposed storage target	CWE-200	upload_to_cloud()
	AWS Access Key printed to logs on error	Critical	Credential leakage	CWE-532	upload_to_cloud()
	Hardcoded SMTP credential	High	Credentials stored in code	CWE-798	send_notification_email()
	No validation for webhook payload	Critical	Blind trust in user input	CWE-915 / OWASP A5 Broken Validation	process_webhook_data()
	Webhook forwarded via HTTP (not HTTPS)	High	Data can be intercepted	CWE-319	WEBHOOK_ENDPOINT
	Arbitrary user deletion via webhook	Critical	No authorization check	CWE-285 Broken Access Control	process_webhook_data()
	No	Critical	Anyone can	CWE-306	process_webhook_data

	authentication for webhook endpoint		invoke destructive actions		0
	Verbose error messages reveal sensitive info	Medium	Error messages expose stack/credentials	CWE-209	Multiple functions

2. Detailed Explanations

➤ 1 .Hardcoded API Key

Description:

The API key is embedded directly in the source code. Anyone with code access can extract it.

Impact:

Attackers can impersonate the service, make unauthorized API calls, or incur financial charges.

Evidence:

```
API_KEY = "sk-1234567890abcdef1234567890abcdef"
```

Fix:

Use environment variables:

```
API_KEY = os.getenv("API_KEY")
```

Verification:

Ensure running the program without the environment variable causes a safe failure.

➤ 2.Hardcoded Database Password

Description:

Database password is stored in plaintext inside the source.

Impact:

Leads to database compromise, unauthorized queries, or full data exfiltration.

Evidence:

```
DATABASE_PASSWORD = "admin123"
```

Fix:

Move credentials to environment variables or a vault (AWS Secrets Manager).

Verification:

Run the program without code changes on a machine where secrets are stored externally.

➤ 3.Hardcoded AWS Access Keys

Description:

AWS Access Key/Secret Key appear directly in the code.

Impact:

This allows full AWS account takeover, S3 deletion, or mass data theft.

Evidence:

```
AWS_ACCESS_KEY = "AKIAIOSFODNN7EXAMPLE"
```

```
AWS_SECRET_KEY = "wJalrXUtnFEMI..."
```

Fix:

Use IAM roles or environment variables. Rotate exposed keys immediately.

Verification:

Run aws sts get-caller-identity using the program; it should show IAM role, not hardcoded user keys

➤ 4. Hardcoded SMTP Password

Description:

Email password is visible in the code.

Impact:

Allows email takeover, phishing attacks, or mass spam.

Evidence:

```
SMTP_PASSWORD = "email_password_123"
```

Fix:

Use environment variables or secrets manager.

Verification:

Ensure email function works with secrets loaded from secure storage.

➤ **5. Logging Sensitive Secrets**

Description:

The system logs credentials, exposing them to log files.

Impact:

Any log reader can steal credentials and compromise systems.

Evidence:

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

Fix:

Remove sensitive info from logs.

Verification:

Search application logs for leaked secrets — none should appear.

➤ **6. TLS Certificate Verification Disabled**

Description:

HTTP requests disable certificate validation (verify=False).

Impact:

Allows Man-in-the-Middle (MITM) attacks.

Evidence:

```
self.session.verify = False
```

Fix:

Remove verify=False and use HTTPS with proper CA bundles.

Verification:

Run API calls and confirm they fail if connecting to an invalid certificate.

➤ 7. SQL Injection

Description:

SQL query constructed unsafely using string interpolation.

Impact:

Attackers can run arbitrary SQL, delete data, or dump database.

Evidence:

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

Fix:

Parameterized SQL:

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

Verification:

Attempt payload: user_id="1 OR 1=1" → query must not execute.

➤ 8. Storing Plaintext Passwords

Description:

User passwords are stored unencrypted.

Impact:

If DB is breached, all accounts are compromised.

Evidence:

password TEXT

Fix:

Hash passwords (bcrypt, Argon2).

Verification:

Check DB — stored values should not match user input.

➤ 9. Storing Credit Cards Unencrypted

Description:

Credit card numbers are stored in plaintext.

Impact:

Severe PCI DSS violation; financial fraud.

Evidence:

credit_card TEXT

Fix:

Encrypt using AES-256 or tokenize.

Verification:

Inspect DB — values must be encrypted.

10. Storing SSNs Unencrypted

Description:

Sensitive personally identifiable information left unprotected.

Impact:

Identity theft and legal liability.

Evidence:

ssn TEXT

Fix:

Encrypt or redact.

Verification:

Check stored values — must be unreadable.

11. Database Connection String Leaked in Logs

Description:

Error logs print the connection string including password.

Impact:

Credentials leak via logs.

Evidence:

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

Fix:

Remove sensitive fields from logs.

Verification:

Trigger DB error → logs should not contain passwords.

12. No Rate Limiting on External API

Description:

Application can spam the external API.

Impact:

DoS risk, cost spikes.

Evidence:

```
response = self.session.post(...)
```

Fix:

Use throttling / retry controls.

Verification:

Send rapid requests; throttler should prevent overload.

13. API Requests Without TLS Validation

Description:

API calls with disabled verification.

Impact:

MITM attack, response tampering.

Evidence:

```
verify=False
```

Fix:

Enable TLS validation.

Verification:

Requests should fail against invalid TLS endpoints.

14. AWS S3 Upload Uses Hardcoded Credentials

Description:

AWS keys embedded in code used for uploads.

Impact:

S3 bucket compromise and data theft.

Evidence:

```
boto3.client(... aws_access_key_id=AWS_ACCESS_KEY ...)
```

Fix:

Use IAM roles. Remove hardcoded keys.

Verification:

Upload file → CloudTrail should show role access, not key.

 **15. Sensitive S3 Bucket Name Exposed****Description:**

Bucket name discloses sensitive operational info.

Impact:

Attackers know where sensitive files are stored.

Evidence:

```
bucket_name="company-sensitive-data"
```

Fix:

Use environment variable, rename bucket.

Verification:

Code should not reveal bucket name.

 **16. AWS Key Leaked in Logs****Description:**

Error logs include AWS access key.

Impact:

Credentials fully exposed.

Evidence:

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

Fix:

Remove exposure of credentials in logs.

Verification:

Break upload → logs must not show credentials.

 **17. Hardcoded SMTP Credentials****Description:**

Email password stored inline.

Impact:

Email account compromise.

Evidence:

```
server.login(sender_email, SMTP_PASSWORD)
```

Fix:

Use environment variables.

Verification:

Send test email after secrets moved to environment.

18. No Webhook Payload Validation

Description:

Webhook accepts arbitrary unvalidated input.

Impact:

Attackers can trigger actions without constraints.

Evidence:

```
user_id = webhook_data.get('user_id')
```

Fix:

Validate schema using Pydantic or custom validator.

Verification:

Send malformed webhook → should reject request.

19. Webhook Forwarded Over HTTP

Description:

Request sent to insecure internal service.

Impact:

MITM inside internal network.

Evidence:

WEBHOOK_ENDPOINT = <http://internal-webhook.company.com>

Fix:

Use HTTPS.

Verification:

Requests to insecure HTTP endpoint should fail.

 **20. Arbitrary User Deletion via Webhook****Description:**

Webhook includes destructive DB action with no authorization.

Impact:

Remote attacker can delete accounts.

Evidence:

```
query = f"DELETE FROM user_data WHERE id = {user_id}"
```

Fix:

Use RBAC checks + parameterized SQL.

Verification:

Unauthorized webhook must not delete users.

 **21. No Authentication for Webhook Endpoint****Description:**

Any actor can trigger critical logic.

Impact:

Unrestricted account deletion / data changes.

Evidence:

Nothing checks tokens, signatures, headers.

Fix:

Use HMAC signature validation or JWT.

Verification:

Webhook without signature must fail.

 **22. Verbose Error Messages****Description:**

Errors leak internal details such as stack traces and credentials.

Impact:

Info disclosure aiding attackers.

Evidence:

```
self.logger.error(f"Query failed: {e}")
```

Fix:

Use generic error messages; send details to secure logs only.

Verification:

Trigger errors — ensure user-facing output does not expose sensitive info.

SECTION 3 — Secure Refactored Code

 **Remove Hardcoded API Key**

 **Insecure**

```
API_KEY = "sk-1234567890abcdef1234567890abcdef"
```

 **Secure**

```
API_KEY = os.getenv("API_KEY")
```

```
if not API_KEY:
```

```
    raise RuntimeError("API_KEY missing from environment")
```

 **Remove Hardcoded Database Password**

 **Insecure**

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

 **Secure**

```
DB_CONNECTION_STRING = os.getenv("DATABASE_URL")
```

 **Remove Hardcoded AWS Keys (Use IAM Role)**

 **Insecure**

```
s3_client = boto3.client(  
    's3',  
    aws_access_key_id=AWS_ACCESS_KEY,  
    aws_secret_access_key=AWS_SECRET_KEY  
)
```

 **Secure**

```
s3_client = boto3.client('s3') # IAM role automatically used
```

 **4. Secure SMTP Credentials**

 **Insecure**

```
server.login(sender_email, SMTP_PASSWORD)
```

 **Secure**

```
SMTP_PASSWORD = os.getenv("SMTP_PASSWORD")
```

```
server.login(sender_email, SMTP_PASSWORD)
```

 **5. Remove Sensitive Logging**

 **Insecure**

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

 **Secure**

```
self.logger.info("Processor initialized successfully.")
```

 **6. Enforce HTTPS + Certificate Validation**

 **Insecure**

```
self.session.verify = False
```

```
urllib3.disable_warnings(...)
```

 **Secure**

```
self.session.verify = True
```

7. Fix SQL Injection

✗ Insecure

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

```
cursor.execute(query)
```

✓ Secure

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

8. Hash Passwords Instead of Plaintext

✗ Insecure

```
password TEXT
```

✗ Insecure insert

```
cursor.execute("INSERT INTO user_data (username, password) VALUES (?, ?)", (u, p))
```

✓ Secure

```
import bcrypt
```

```
hashed = bcrypt.hashpw(password.encode(), bcrypt.gensalt())
```

```
cursor.execute("INSERT INTO user_data (username, password) VALUES (?, ?)",  
(username, hashed))
```

9. Encrypt Credit Cards

✗ Insecure

```
credit_card TEXT
```

✓ Secure

```
from cryptography.fernet import Fernet
```

```
f = Fernet(os.getenv("ENCRYPTION_KEY"))
```

```
encrypted = f.encrypt(card_number.encode())
```

10. Encrypt SSNs

Insecure

ssn TEXT

Secure

Same encryption pattern as credit card.

11. Remove DB Connection Leaks

Insecure

```
self.logger.error(f"Database failed: {e} | Connection: {DB_CONNECTION_STRING}")
```

Secure

```
self.logger.error("Database connection failed.")
```

12. Add Rate Limiting

Insecure

```
self.session.post(...)
```

Secure

```
from requests.adapters import HTTPAdapter
```

```
from urllib3.util.retry import Retry
```

```
retry = Retry(total=5, backoff_factor=0.3, status_forcelist=[429, 500, 502, 503])
```

```
self.session.mount("https://", HTTPAdapter(max_retries=retry))
```

13. Enforce API TLS


```
verify=False
```



```
verify=True
```

 **14. Secure S3 Upload (No Keys)**

✗

Hardcoded AWS keys.

✓

```
s3_client = boto3.client('s3') # IAM role  
s3_client.upload_file(file_path, bucket_name, os.path.basename(file_path))
```

 **15. Hide Sensitive Bucket Name**

✗

```
bucket_name="company-sensitive-data"
```

✓

```
bucket_name = os.getenv("UPLOAD_BUCKET")
```

 **16. Remove AWS Key from Error Logs**

✗

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

✓

```
self.logger.error("S3 upload failed.")
```

 **17. Secure SMTP Credentials**

Already covered in #4.

 **18. Validate Webhook Payload**

✗ **Insecure**

```
user_id = webhook_data.get('user_id')
```

✓ **Secure**

```
from pydantic import BaseModel
```

```
class Webhook(BaseModel):  
  
    user_id: int  
  
    action: str  
  
  
payload = Webhook(**webhook_data)
```

 **19. Use HTTPS for Webhook Forward**

✗

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

✓

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

 **20. Prevent Arbitrary User Deletion**

✗

```
if action == 'delete_user':
```

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

✗ **No authentication.**

✓ **Secure (RBAC + parameterized SQL)**

```
if action == "delete_user" and user_is_authorized(request):
```

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

 **21. Add Webhook Authentication**

✗

No token/signature.

✓

```
signature = request.headers.get("X-Signature")
```

```
verify_signature(signature, request.body)
```

Using:

- HMAC SHA-256
- Shared secret
- **22. Remove Verbose Error Messages**



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



self.logger.error("Operation failed.")

SECTION 4 — Verification & Security Testing Plan

This section describes how to verify that each remediation implemented in Section 3 is working and secure. The goal is to ensure that all identified vulnerabilities are fully resolved and that no regressions occur.

- **4.1 Secrets & Credentials Verification**
- **Test 1 — Environment Variables Loaded Correctly**

Steps:

1. Remove API_KEY, DB_PASSWORD, SMTP_PASSWORD values from the code.
2. Export them in your shell:

```
export API_KEY="test-key"
```

3. Run the application.

Expected Result:

- Application starts normally.
- If missing, it fails with a safe error:
“API_KEY missing from environment”

➤ **Test 2 — Secrets Not Logged**

Steps:

1. Review application logs after startup.
2. Trigger all major functions (email, S3, database).
3. Search logs for sensitive data:

```
grep -Ei '(key|password|secret)' app.log
```

Expected Result:

- No secrets appear.
- Only safe operational logs exist.

➤ **4.2 SQL Injection Verification**

Test Input:

Attempt calling:

```
fetch_user_data("1 OR 1=1")
```

Expected Result:

- Function should raise validation error OR return no results.
- Database should NOT return all rows (attack failed).

Automation Test:

```
assert fetch_user_data("1 OR 1=1") is None
```

➤ **4.3 Database Encryption Verification**

✓ **Test 1 — Password Hash Verification**

Steps:

1. Register a user with password “admin123”.
2. Inspect the stored password in the DB.

Expected Result:

- Password is hashed (bcrypt/argon2).
- Stored value does not match plaintext.
- Hash value changes each registration (due to salt).

✓ **Test 2 — Credit Card & SSN Encryption**

Steps:

1. Store a fake credit card & SSN.
2. Inspect the record in DB.

Expected Result:

- Value is encrypted (Fernet AES-256).
- Value is unreadable.
- Decryption only possible through your service.

➤ **4.4 TLS & HTTPS Verification**

✓ **Test 1 — Certificate Validation**

Trigger an API call to an endpoint with an invalid certificate.

Expected Result:

- Request fails safely.
- No verify=False exists in code.

✓ **Test 2 — Webhook HTTPS Enforcement**

Send simulated webhook forwarding to HTTP.

Expected Result:

- Application refuses to send data to insecure URL.
- Logs warn “insecure endpoint”.

➤ **4.5 Cloud Verification (AWS S3)**

✓ **Test 1 — IAM Role Validation**

Run:

```
aws sts get-caller-identity
```

Expected Result:

- Caller identity is an IAM role (EC2/ECS/Lambda).
- Not an Access Key.

✓ Test 2 — S3 Upload Attempt

Upload a file via the application.

Expected Result:

- File uploads successfully.
- No AWS key found in logs.
- CloudTrail confirms access via role.

➤ 4.6 Webhook Authorization Verification

✓ Test 1 — Missing Signature

Send a webhook request with no signature header.

Expected Result:

- Application rejects request (HTTP 401 or 403).
 - No destructive action occurs.
- ✓ Test 2 — Incorrect Signature**

Use wrong HMAC signature.

Expected Result:

- Request is rejected.
 - Logs show safe message “Invalid signature”.
- ✓ Test 3 — Unauthorized Deletion Attempt**

Send:

```
{
```

```
"user_id": 1,  
"action": "delete_user"  
}
```

Expected Result:

- Application checks authorization.
 - Deletion does **not** occur unless authorized.
- **4.7 Error Handling Verification**
✓ **Test 1 — Trigger errors**

Break the DB connection or S3 upload.

Expected Result:

- Logs contain generic messages only (“Operation failed”)
- No stack trace or secrets are leaked.

➤ **4.8 Rate Limiting & Retry Logic Verification**

Test Steps:

1. Flood the API endpoint with rapid requests.
2. Examine service behavior.

Expected Result:

- Requests are throttled.
- Exponential backoff functions correctly.
- No DoS or unbounded retries occur.

➤ **4.9 Static Analysis & Security Scanning**

Tools to Run:

- **Bandit** (Python SAST):
- bandit -r .

- **Semgrep** (policy rules):
 - semgrep --config auto .
- **trivy fs** (secrets & misconfig):
 - trivy fs .

Expected Result:

- Zero “High” or “Critical” findings.
- No hardcoded credentials detected.

➤ 4.10 Penetration Test Checklist

Confirm the following:

- No SQL injection
- No XSS in logs or webhook payload
- HTTPS enforced globally
- Auth required for sensitive actions
- Encryption used for all sensitive data
- No insecure deserialization
- No open S3 buckets
- No leaked credentials in Git history

SECTION 5 — Security Best Practices Summary

This section outlines the high-level, long-term security principles that the application and development team should follow to prevent recurrence of similar vulnerabilities. It complements the fixes by establishing durable safeguards across infrastructure, coding practices, and operational processes.

5.1 Secrets & Credentials Management

Best Practices

- Use a centralized secrets manager:

- AWS Secrets Manager
- HashiCorp Vault
- GCP Secret Manager
- Never store credentials in:
 - Source code
 - Git history
 - Log files
 - Configuration files checked into version control
- Rotate all secrets regularly (30–90 days).
- Enforce MFA for all cloud dashboard access.
- Use short-lived tokens instead of long-lived keys.

Outcome

Prevents credential leaks, unauthorized access, and cloud account compromise.

5.2 Secure Coding & Input Validation

Best Practices

- Enforce input validation across:
 - Webhooks
 - API request bodies
 - Query parameters
 - Form submissions
- Use strong typing & schema validation (Pydantic / Marshmallow).
- Avoid dynamic SQL or string interpolation.
- Implement Least Privilege (POLP) for DB queries.

Outcome

Prevents SQL injection, unexpected behavior, and input-based attacks.

5.3 Data Protection & Encryption

Best Practices

- Encrypt sensitive data *in transit and at rest*:
 - TLS 1.2/1.3 everywhere
 - AES-256-GCM for database fields
- Never store sensitive data (credit card, SSN) unencrypted.
- Hash all passwords using:
 - Argon2id (preferred)
 - Bcrypt (acceptable)

Outcome

Mitigates identity theft, data breaches, and compliance violations (GDPR, PCI-DSS).

5.4 Cloud Security & IAM

Best Practices

- Use IAM roles instead of access keys.
- Mandate least privilege for all AWS IAM permissions.
- Apply S3 bucket security policies:
 - Block public access
 - Enforce encryption at rest (SSE-S3 or SSE-KMS)
- Enable CloudTrail logging for all access.
- Use security groups and NACLs to restrict network access.

Outcome

Prevents cloud takeover and secures storage, compute, and networking.

5.5 Logging, Monitoring & Incident Detection

Best Practices

- Sanitize all logs:

- No passwords
- No tokens
- No secrets
- No PII
- Use centralized logging:
 - CloudWatch
 - ELK Stack
 - Datadog
- Implement anomaly detection:
 - Unusual IPs
 - High request volumes
 - Repeated failed requests
 - Suspicious deletion attempts
- Set up automated alerts for:
 - Failed logins
 - Disabled TLS
 - Permission-denied actions

Outcome

Provides early detection of attacks and compliance visibility.

5.6 Secure Deployment / DevOps Practices

Best Practices

- Shift-left security:
 - Run Bandit, Semgrep in CI
 - Scan dependencies with Trivy or Snyk
- Enforce branch protection rules.

- Require code reviews for all security-sensitive changes.
- Scan Docker images for vulnerabilities.
- Use Infrastructure-as-Code (IaC) scanning (Checkov, Terraform scan).

Outcome

Prevents vulnerabilities from entering production.

5.7 Network Security

Best Practices

- Enforce HTTPS across all services.
- Disable any insecure protocols:
 - HTTP
 - TLS < 1.2
- Use firewalls/WAFs to block attacks.
- Restrict outbound egress only to trusted endpoints.
- Use subnet isolation for:
 - Databases
 - Internal APIs
 - Admin services

Outcome

Provides strong defense against MITM, eavesdropping, and network-based attacks.

5.8 Operational Security & Governance

Best Practices

- Regularly train developers on secure coding.
- Maintain an up-to-date security policy.
- Enforce least privilege across:
 - Developers

- Admins
- Service accounts
- Maintain audit logs for:
 - User actions
 - Sensitive actions
 - Webhook commands
- Conduct periodic penetration tests.

Outcome

Improves organizational readiness and security posture.