Passflask: Test cases and static analyses

Test Cases

These tests **mock** database interactions and file reads with `unittest.mock.patch` to keep tests **fast** and **isolated** from external services. We mock or patch `check_leaks` to avoid reading from the actual file.

*When testing JWT logic, I aim to confirm that **encoding** and **decoding** are done with the same secret and algorithm. JWT operations are mocked in these unit tests to focus on verifying the application's *response handling*

Each test generates and checks **status codes** and **error messages**.

The tests are **functional** rather than performance-oriented.

```
import os
import pytest
import json
from unittest.mock import patch, MagicMock
from app import app

@pytest.fixture
def client():
    """
    Pytest fixture to create a test client from the Flask app.
    """
    app.config["TESTING"] = True
    with app.test_client() as client:
        yield client

# ------------------------------------------------------------------------
# /generate endpoint tests
# ------------------------------------------------------------------------

@patch("app.check_leaks", return_value=False)
@patch("app.upload_to_db", return_value=None)
def test_generate_valid(mock_upload_to_db, mock_check_leaks, client):
    """
    Test a valid POST /generate request.
    Should return a JSON response with 'generated_password' and 'token'.
    """
    payload = {"username": "testuser"}
    response = client.post("/generate", json=payload)

    assert response.status_code == 200, "Expected 200 OK for valid request."
    data = response.get_json()
    assert "generated_password" in data, "Response should contain a
generated_password."
    assert "token" in data, "Response should contain a JWT token."

    # Ensure our mocks were called
    mock_check_leaks.assert_called_once()
    mock_upload_to_db.assert_called_once()
```

```
@patch("app.check_leaks", side_effect=[True, False])
@patch("app.upload_to_db", return_value=None)
def test_generate_leaked_password(mock_upload_to_db, mock_check_leaks, client):
    """
    Test that if a generated password is leaked, the service discards it and
generates a new one.
    The first call to check_leaks returns True (leaked), the second call
returns False.
    """
    payload = {"username": "testuser"}
    response = client.post("/generate", json=payload)

    assert response.status_code == 200
    data = response.get_json()
    assert "generated_password" in data
    assert "token" in data

    # check_leaks should have been called twice because the first password is
leaked.
    assert mock_check_leaks.call_count == 2

def test_generate_missing_username(client):
    """
    Test /generate endpoint with missing username in the JSON payload.
    Should return 400 with an error message.
    """
    response = client.post("/generate", json={})

    assert response.status_code == 400, "Expected 400 Bad Request when username
is missing."
    data = response.get_json()
    assert "error" in data, "Response should contain an error message."

def test_generate_no_json(client):
    """
    Test /generate endpoint with no JSON body.
    Should return 400 with an error message.
    """
    response = client.post("/generate")

    assert response.status_code == 400, "Expected 400 Bad Request when request
body is not JSON."
    data = response.get_json()
    assert "error" in data, "Response should contain an error message."

# ----------------------------------------------------------------------------
# /retrieve endpoint tests
# ----------------------------------------------------------------------------

@patch("app.generate_uid_hash", return_value="fake_uid_hash")
@patch("app.jwt.decode")
@patch("app.jwt_gen", return_value="new_jwt_token")
@patch("app.mysql.connector.connect")
def test_retrieve_valid(
    mock_connect,
```

```
    mock_jwt_gen,
    mock_jwt_decode,
    mock_generate_uid_hash,
    client
):
    """
    Test a valid /retrieve request with correct username and valid token.
    Should return 200 and a new JWT token.
    """
    # Mock database query results
    mock_cursor = MagicMock()
    mock_cursor.fetchone.return_value = [1]  # Means user record found
    mock_connect.return_value.cursor.return_value = mock_cursor

    # Mock JWT decode success
    mock_jwt_decode.return_value = {
        "sub": "fake_uid_hash",
        "ist": "2025-01-20T10:00:00",
        "exp": "9999-01-01T00:00:00"  # far future
    }

    # Make request
    response = client.get(
        "/retrieve",
        headers={"Authorization": "valid_jwt_token"},
        query_string={"username": "testuser"}
    )

    assert response.status_code == 200
    data = response.get_json()
    assert "new_token" in data
    assert data["new_token"] == "new_jwt_token"

@patch("app.generate_uid_hash", return_value="fake_uid_hash")
@patch("app.jwt.decode", side_effect=Exception("Invalid token"))
@patch("app.mysql.connector.connect")
def test_retrieve_invalid_token(
    mock_connect,
    mock_jwt_decode,
    mock_generate_uid_hash,
    client
):
    """
    Test /retrieve with an invalid token. Should return 401 with an error
message.
    """
    mock_cursor = MagicMock()
    mock_cursor.fetchone.return_value = [1]
    mock_connect.return_value.cursor.return_value = mock_cursor

    response = client.get(
        "/retrieve",
        headers={"Authorization": "invalid_jwt_token"},
        query_string={"username": "testuser"}
    )
    assert response.status_code == 401
```

```
    data = response.get_json()
    assert "error" in data

@patch("app.mysql.connector.connect")
def test_retrieve_no_token(mock_connect, client):
    """
    Test /retrieve with no token in the Authorization header. Should return
400.
    """
    response = client.get(
        "/retrieve",
        query_string={"username": "testuser"}  # missing token
    )
    assert response.status_code == 400
    data = response.get_json()
    assert "error" in data

@patch("app.mysql.connector.connect")
def test_retrieve_no_username(mock_connect, client):
    """
    Test /retrieve with no username in the query string. Should return 400.
    """
    response = client.get(
        "/retrieve",
        headers={"Authorization": "valid_jwt_token"}
    )
    assert response.status_code == 400
    data = response.get_json()
    assert "error" in data

@patch("app.generate_uid_hash", return_value="fake_uid_hash")
@patch("app.jwt.decode", return_value={
    "sub": "fake_uid_hash",
    "ist": "2025-01-20T10:00:00",
    "exp": "2025-01-19T10:00:00"  # expired
})
@patch("app.mysql.connector.connect")
def test_retrieve_expired_token(
    mock_connect,
    mock_jwt_decode,
    mock_generate_uid_hash,
    client
):
    """
    Test /retrieve with an expired token. Should return 401 with 'Token has
expired'.
    """
    mock_cursor = MagicMock()
    mock_cursor.fetchone.return_value = [1]
    mock_connect.return_value.cursor.return_value = mock_cursor

    response = client.get(
        "/retrieve",
        headers={"Authorization": "expired_jwt_token"},
        query_string={"username": "testuser"}
    )
```

```
    assert response.status_code == 401
    data = response.get_json()
    assert "error" in data
    assert "expired" in data["error"]

@patch("app.generate_uid_hash", return_value="fake_uid_hash")
@patch("app.jwt.decode", return_value={
    "sub": "some_other_uid_hash",
    "ist": "2025-01-20T10:00:00",
    "exp": "2025-01-21T10:00:00"
})
@patch("app.mysql.connector.connect")
def test_retrieve_sub_mismatch(
    mock_connect,
    mock_jwt_decode,
    mock_generate_uid_hash,
    client
):
    """
    Test /retrieve where the token's sub claim does not match the username's
uid_hash.
    Should return 401 Invalid token.
    """
    mock_cursor = MagicMock()
    mock_cursor.fetchone.return_value = [1]
    mock_connect.return_value.cursor.return_value = mock_cursor

    response = client.get(
        "/retrieve",
        headers={"Authorization": "mismatched_jwt_token"},
        query_string={"username": "testuser"}
    )
    assert response.status_code == 401
    data = response.get_json()
    assert "error" in data
    assert "Invalid token" in data["error"]

@patch("app.generate_uid_hash", return_value="nonexistent_uid_hash")
@patch("app.jwt.decode", return_value={
    "sub": "nonexistent_uid_hash",
    "ist": "2025-01-20T10:00:00",
    "exp": "2025-01-21T10:00:00"
})
@patch("app.mysql.connector.connect")
def test_retrieve_invalid_username_in_db(
    mock_connect,
    mock_jwt_decode,
    mock_generate_uid_hash,
    client
):
    """
    Test /retrieve where the username is hashed to something that does not
exist in the DB.
    Should return 404.
    """
    mock_cursor = MagicMock()
```

```
    mock_cursor.fetchone.return_value = [0]  # no user found
    mock_connect.return_value.cursor.return_value = mock_cursor

    response = client.get(
        "/retrieve",
        headers={"Authorization": "valid_jwt_token"},
        query_string={"username": "invalid_user"}
    )
    assert response.status_code == 404
    data = response.get_json()
    assert "error" in data
    assert "Invalid username" in data["error"]
```

I am going to explain 5 tests: the most important ones corresponding to core, measurable functionality.

1. `test_generate_valid`

```
pythonCopy@patch("app.check_leaks", return_value=False)
@patch("app.upload_to_db", return_value=None)
def test_generate_valid(mock_upload_to_db, mock_check_leaks, client):
    ...
```

1. **What It Tests**:
    - This test ensures that a valid POST request to the `/generate` endpoint **successfully** creates a password and returns a **JWT token**.
    - It checks the application's **happy path**: the input is correct, the generation process is smooth, no leaks, and uploading to the database is successful.
2. **Key Steps**:
    - **Mocking** **`check_leaks`** to always return `False`, simulating a scenario where the generated password is *not* in the leaked passwords list.
    - **Mocking** **`upload_to_db`** to avoid making real database calls during this unit test.
    - **Calling** the `/generate` endpoint with a JSON payload containing a valid `"username": "testuser"`.
    - **Verifying** that the response contains both `"generated_password"` and `"token"` in the JSON body.
    - **Asserting** that the HTTP status code is `200 OK`.
3. **Pass/Fail Criteria**:
    - **Pass**: The service responds with `200 OK`, and the response body includes keys `"generated_password"` and `"token"`.
    - **Fail**: If the endpoint returns an error status code (e.g., 400 or 500), or if the JSON response does not contain the expected fields.

2. `test_generate_leaked_password`

```
pythonCopy@patch("app.check_leaks", side_effect=[True, False])
@patch("app.upload_to_db", return_value=None)
def test_generate_leaked_password(mock_upload_to_db, mock_check_leaks, client):
    ...
```

1. **What It Tests**:
    - Validates the logic that *discards* a leaked password and *re-generates* a new one.
    - Specifically tests the **loop**: if the first password is found in the leaked passwords list, the app should generate another password.
2. **Key Steps**:
    - **Mocking** **`check_leaks`** to first return `True` (the generated password is leaked), then return `False` on the second call (the new password is safe).
    - **Calling** `/generate` with a valid username.
    - **Expecting** the final response still includes `"generated_password"` and `"token"`.
3. **Pass/Fail Criteria**:
    - **Pass**: The test ends with a `200 OK`, and the final generated password is not leaked. Also, `check_leaks` must be called *twice*.
    - **Fail**: If the app does not retry password generation after the first leak, or if it fails to return the expected JSON structure.

3. `test_retrieve_valid`

```
pythonCopy@patch("app.generate_uid_hash", return_value="fake_uid_hash")
@patch("app.jwt.decode")
@patch("app.jwt_gen", return_value="new_jwt_token")
@patch("app.mysql.connector.connect")
def test_retrieve_valid(...):
    ...
```

1. **What It Tests**:
    - The successful retrieval of a new JWT token when the client provides:
        1. A valid `username` that exists in the database.
        2. A valid JWT token that has not expired and has the correct `sub` claim.
2. **Key Steps**:
    - **Mocking** **`mysql.connector.connect()`** to simulate a database where `uid_hash` is found (`fetchone()` returns `[1]`).
    - **Mocking** **`jwt.decode`** to avoid real JWT verification complexities (the mock returns a payload indicating a valid token).
    - **Mocking** **`jwt_gen`** to always return `"new_jwt_token"`.
    - **Verifying** the response is `200` with a JSON key `"new_token"`.
3. **Pass/Fail Criteria**:

- **Pass**: The service responds with `200` and returns `{"new_token": "new_jwt_token"}`.
- **Fail**: If any of these conditions are not met or a different HTTP status is returned.

4. `test_retrieve_sub_mismatch`

```
pythonCopy@patch("app.jwt.decode", return_value={
    "sub": "some_other_uid_hash",
    "ist": "...",
    "exp": "2025-01-21T10:00:00"
})
def test_retrieve_sub_mismatch(...):
    ...
```

1. **What It Tests**:
   - Ensures the token's `sub` claim matches the user's hashed username (`uid_hash`).
   - If `sub` does not match the actual user, the server must reject the request (`401`).
2. **Key Steps**:
   - **Mocking** `jwt.decode` to produce a mismatch between `sub` and the real `uid_hash`.
   - **Expecting** the endpoint to respond with `401` and an "Invalid token" error.
3. **Pass/Fail Criteria**:
   - **Pass**: The service denies access (`401`) and returns an error message.
   - **Fail**: If it issues a new token despite the mismatch.

5. `test_retrieve_invalid_username_in_db`

```
pythonCopy@patch("app.mysql.connector.connect")
def test_retrieve_invalid_username_in_db(...):
    ...
```

1. **What It Tests**:
   - The logic that checks whether the user exists in the database.
   - If the `uid_hash` is not found, the endpoint responds with `404 Not Found`.
2. **Key Steps**:
   - **Mocking** the DB cursor to return `[0]` for the `COUNT(*)` query, indicating no matching records.
   - **Expecting** a `404` with `"Invalid username"`.
3. **Pass/Fail Criteria**:
   - **Pass**: The endpoint rejects the request with `404`.
   - **Fail**: If it proceeds with token verification despite the user not existing in the DB.

Static analysis results

Bandit

```
$ bandit app.py
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.13.1
Run started:2025-01-20 20:19:14.276648

Test results:
>> Issue: [B311:blacklist] Standard pseudo-random generators are not suitable
for security/cryptographic purposes.
   Severity: Low   Confidence: High
   CWE: CWE-330 (https://cwe.mitre.org/data/definitions/330.html)
   More Info:
https://bandit.readthedocs.io/en/1.8.2/blacklists/blacklist_calls.html#b311-
random
   Location: ./app.py:56:23
55          char_pool = list(char_pool)
56          password = ''.join(random.choice(char_pool) for _ in
range(PASSWORD_POLICY["length"]))
57          return password

--------------------------------------------------
>> Issue: [B201:flask_debug_true] A Flask app appears to be run with
debug=True, which exposes the Werkzeug debugger and allows the execution of
arbitrary code.
   Severity: High   Confidence: Medium
   CWE: CWE-94 (https://cwe.mitre.org/data/definitions/94.html)
   More Info:
https://bandit.readthedocs.io/en/1.8.2/plugins/b201_flask_debug_true.html
   Location: ./app.py:185:4
184     if __name__ == "__main__":
185         app.run(debug=True)

--------------------------------------------------

Code scanned:
        Total lines of code: 133
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0
                Low: 1
                Medium: 0
                High: 1
        Total issues (by confidence):
                Undefined: 0
                Low: 0
                Medium: 1
                High: 1
```

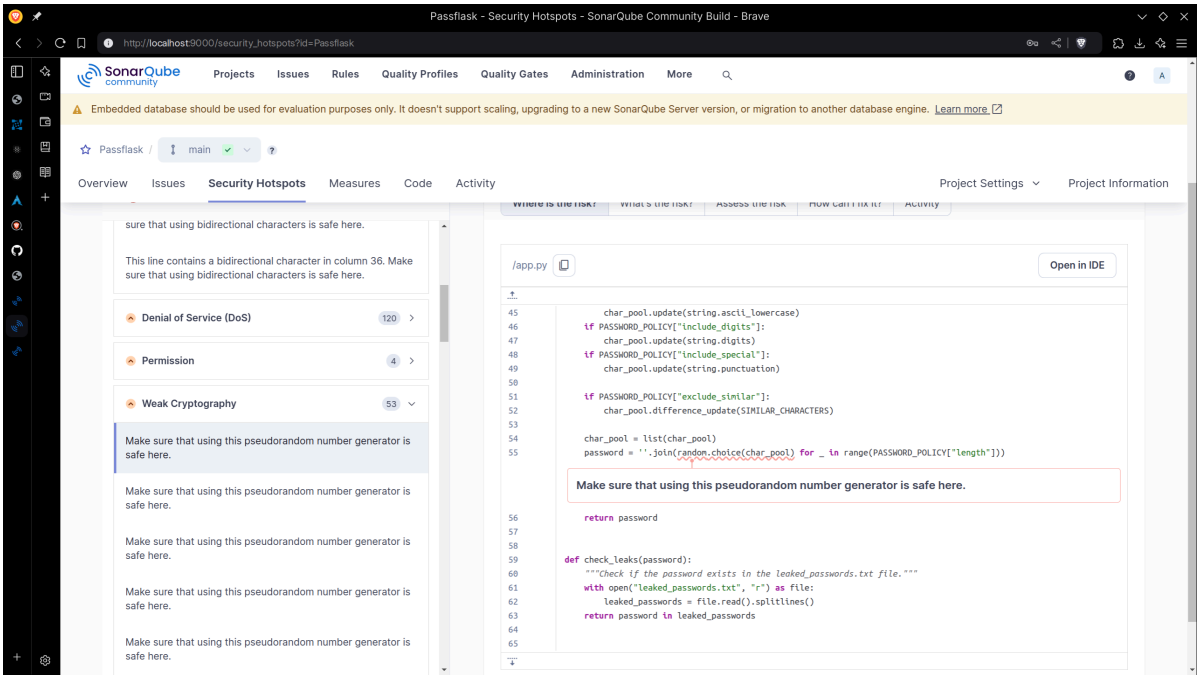
```
Files skipped (0):
```

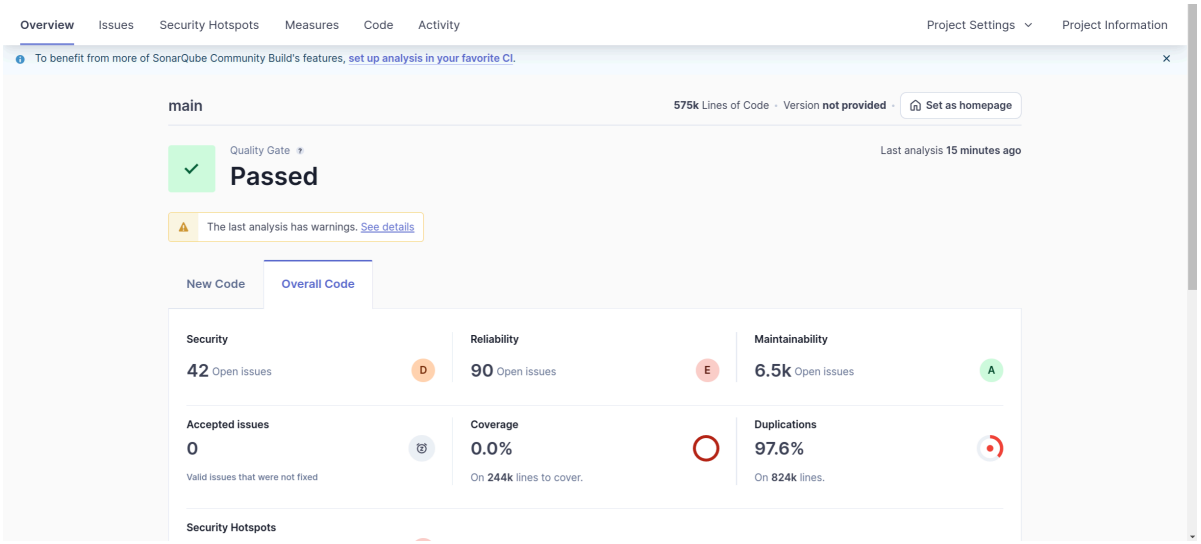
Summary

1. **Issue [B311: blacklist] - Pseudo-Random Generator**
   - **Location**: `app.py`, line 56
   - **Severity**: Low | **Confidence**: High
   - **Details**: Using `random.choice()` for password generation is flagged because the `random` module is not considered cryptographically secure.
   - **Recommendation**: Replace `random.choice()` with a cryptographically secure method such as `secrets.choice()` (from the `secrets` module).
2. **Issue [B201: flask_debug_true] - Debug Mode in Production**
   - **Location**: `app.py`, line 185
   - **Severity**: High | **Confidence**: Medium
   - **Details**: Running the Flask application with `debug=True` exposes the interactive debugger and can allow the execution of arbitrary code in production environments.
   - **Recommendation**: Disable debug mode when deploying to production (set `debug=False` or remove `debug` entirely).

Sonarqube

The analysis revealed numerous potential vulnerabilities and security suggestions. While an exportable form was not available, the platform's self-hosted web interface provides a server connector and analytics.





Even with this result, static code analysis is just one part of the picture.