Testing report

Table of content

1. Introduction

The purpose of this test suite is to assess the functionality of different parts that might be utilized in a port scanner program. The program, which is most likely built using the port_scanner package, offers capabilities for finding devices on a network, scanning ports on target hosts, and possibly even executing sophisticated operations like TCP SYN scans. These tests build a set of test cases covering important facets of the behavior of the application using the pytest framework. The purpose of the test cases is to verify that the functions from the port_scanner.networking and port_scanner.decorators modules perform as anticipated in a range of circumstances.

Areas Covered by the Tests

- Validating User Input:
  - Tests ensure proper validation of IP addresses, port numbers, and input types for functions like is_ip_address, is_port_open, arp_scan, and tcp_syn_scan.
- Network Reachability and Port Status:
  - Tests verify the ability to ping reachable hosts (ping), identify open and closed ports (is_port_open), and potentially perform advanced scanning techniques like TCP SYN scans (tcp_syn_scan).
- Error Handling:
  - Tests confirm that the functions handle unexpected inputs, network errors, and invalid responses gracefully, raising appropriate exceptions or returning expected values.
- Functionality of Helper Functions:
  - Tests explore the behavior of helper functions like is_ip_address for IP address validation and the rate_limit decorator for controlling function execution frequency.
- Device Discovery:
  - Tests validate the arp_scan function's ability to discover devices within a specified network range.

2. Testing App

```
import pytest
import typer
from port_scanner.app import _typer_check_host, _typer_check_range, app
from typer.testing import CliRunner

runner = CliRunner()
```

```
_LOCALHOST = "127.0.0.1"


def test_app_portscan_localhost_with_open_port(mocker):
    mocker.patch("port_scanner.app.is_port_open", return_value=True)
    mocker.patch("port_scanner.app.ping", return_value=True)

    result = runner.invoke(app, ["port-scan", "--host", f"{_LOCALHOST}", "--start-
port", "20", "--end-port", "20"])
    assert result.exit_code == 0
    assert "20" in result.stdout
    assert "open" in result.stdout
    assert "closed" not in result.stdout


def test_app_portscan_localhost_with_closed_port(mocker):
    mocker.patch("port_scanner.app.is_port_open", return_value=False)
    mocker.patch("port_scanner.app.ping", return_value=True)

    result = runner.invoke(app, ["port-scan", "--host", f"{_LOCALHOST}", "--start-
port", "20", "--end-port", "20"])
    assert result.exit_code == 0
    assert "20" in result.stdout
    assert "closed" in result.stdout
    assert "open" not in result.stdout


def test_app_portscan_localhost_with_multiple_port(mocker):
    mocker.patch("port_scanner.app.is_port_open", return_value=False)
    mocker.patch("port_scanner.app.ping", return_value=True)

    result = runner.invoke(app, ["port-scan", "--host", f"{_LOCALHOST}", "--start-
port", "20", "--end-port", "21"])
    assert result.exit_code == 0
    assert "20" in result.stdout
    assert "21" in result.stdout


def test_app_portscan_ping_failure(mocker):
    mocker.patch("port_scanner.app.ping", return_value=False)

    result = runner.invoke(app, ["port-scan", "--host", f"{_LOCALHOST}", "--start-
port", "20", "--end-port", "20"])
    assert result.exit_code == 1


def test_app_tcp_syn_scan(mocker):
    mocker.patch("port_scanner.app.tcp_syn_scan", return_value=False)
    result = runner.invoke(
        app,
        [
            "port-scan",
```

```
            "--host",
            f"{_LOCALHOST}",
            "--start-port",
            "20",
            "--end-port",
            "20",
            "--use-tcp-syn",
            "--skip-ping",
        ],
    )
    assert result.exit_code == 0


def test_app_arp_scan(mocker):
    mocker.patch("port_scanner.networking.arp_scan", return_value=["10.10.10.10",
"10.1.1.1"])
    result = runner.invoke(app, ["scan-arp", "--ip-range", "10.10.10.0/24"])
    assert result.exit_code == 0


def test_typer_check_host():
    with pytest.raises(typer.BadParameter):
        _typer_check_host("invalid")


def test_typer_check_range():
    with pytest.raises(typer.BadParameter):
        _typer_check_range("invalid")
```

This code establishes a set of tests to verify functionality most likely from a port scanner application (port_scanner.app) using the pytest framework. An explanation of each test's purpose and significance is provided below:

1. Testing port_scanner.app.port_scan function:

- **test_app_portscan_localhost_with_open_port:**
  - Mocks the is_port_open function to always return True (open port).
  - Mocks the ping function to always return True (host reachable).
  - Runs the port-scan command with localhost, port 20 as start and end port.
  - Asserts the exit code is 0 (success), port 20 is mentioned in the output, "open" is found in the output, and "closed" is not present.
  - This test verifies if the application can successfully scan an open port on a reachable host.
- **test_app_portscan_localhost_with_closed_port:**
  - Similar to the previous test, but mocks is_port_open to return False (closed port).
  - Asserts the exit code is 0, port 20 is mentioned, "closed" is found, and "open" is not present.
  - This test verifies if the application can identify a closed port on a reachable host.

- **test_app_portscan_localhost_with_multiple_port:**
    - Mocks is_port_open to return False for both ports.
    - Scans localhost with ports 20 and 21.
    - Asserts the exit code is 0, and both ports (20 and 21) are mentioned with their scan results (likely "closed").
    - This test verifies if the application can scan multiple ports within a range.
- **test_app_portscan_ping_failure:**
    - Mocks ping to return False (host unreachable).
    - Scans localhost with port 20.
    - Asserts the exit code is 1 (failure) since the application might rely on a reachable host for port scanning.
    - This test verifies if the application handles cases where the target host is unreachable.

2. Testing port_scanner.app.tcp_syn_scan **(optional):**

- **test_app_tcp_syn_scan:**
    - Mocks tcp_syn_scan (likely for advanced port scanning) to return False.
    - Runs the command with --use-tcp-syn (TCP SYN scan) and --skip-ping flags.
    - Asserts the exit code is 0, indicating successful execution even if the scan itself might not find open ports.
    - This test verifies if the application can execute a TCP SYN scan (assuming the functionality is implemented) even if it doesn't find open ports.

3. Testing port_scanner.networking.arp_scan **(optional):**

- **test_app_arp_scan:**
    - Mocks arp_scan to return two IP addresses.
    - Runs the scan-arp command with an IP range.
    - Asserts the exit code is 0 and the output includes both discovered IP addresses.
    - This test verifies if the application can utilize the arp_scan function to discover devices within an IP range.

4. Testing helper functions:

- **test_typer_check_host:**
    - Uses pytest.raises to expect a typer.BadParameter exception.
    - Calls _typer_check_host with an invalid host string ("invalid").
    - This test verifies if the helper function for host validation raises an error for invalid input.
- **test_typer_check_range:**
    - Like the previous test but checks the validation of the IP range string.
    - This test verifies if the helper function for IP range validation raises an error for invalid input.

Overall, these tests cover various functionalities of the potential port scanner application, including:

- Scanning open and closed ports on a reachable host
- Handling unreachable hosts
- Potentially using TCP SYN scans
- Discovering devices within an IP range
- Validating user input for host and IP range formats.

3. *Testing Decorators*

```python
# Mock function for testing
import time

from port_scanner.decorators import rate_limit


def test_rate_limit_decorator():
    @rate_limit(interval=2)
    def test_function():
        return "Test"

    # Test that the function is called only once within the interval
    start_time = time.time()
    test_function()
    elapsed_time = time.time() - start_time
    assert elapsed_time < 2

    # Test that the function is called twice when called consecutively
    test_function()
    elapsed_time = time.time() - start_time
    assert elapsed_time > 2


def test_rate_limit_decorator_multiple_functions():
    @rate_limit(interval=2)
    def test_function_1():
        return "Test 1"

    @rate_limit(interval=2)
    def test_function_2():
        return "Test 2"

    # Test that each function respects its own rate limit
    start_time = time.time()
    test_function_1()
    test_function_2()
    elapsed_time = time.time() - start_time
    assert elapsed_time < 2

    # Test that calling both functions consecutively respects the rate limits
    test_function_1()
    test_function_2()
    elapsed_time = time.time() - start_time
    assert elapsed_time > 2
```

These tests verify the functionality of the rate_limit decorator likely found in the port_scanner.decorators module. Here's a breakdown of what each test does and why it's important:

1. test_rate_limit_decorator

- This test defines a function test_function decorated with rate_limit(interval=2). This means the decorated function can only be called once within a 2-second interval.
- It measures the elapsed time between calling test_function twice.
    - The first assertion (elapsed_time < 2) verifies that the second call happens almost immediately, indicating no delay for the first call within the interval.
    - The second assertion (elapsed_time > 2) is executed after the second call to test_function. It ensures that enough time has passed (since the first call, suggesting the decorator enforced a delay before allowing the second call.
- This test ensures the rate_limit decorator successfully restricts the decorated function's execution frequency. It prevents overwhelming the system with too many calls within a short period.

2. test_rate_limit_decorator_multiple_functions

- This test defines two separate functions, test_function_1 and test_function_2, both decorated with rate_limit(interval=2).
- It follows a similar approach to measure elapsed time:
    - The first assertion (elapsed_time < 2) verifies both functions can be called consecutively **without a delay** since they have separate rate limits.
    - The second assertion (elapsed_time > 2) after calling both functions again ensures a **delay of more than 2 seconds** has occurred, indicating the decorator respects the individual rate limit for each function.
- This test verifies that the rate_limit decorator manages rate limits independently for different functions. Each decorated function has its own 2-second interval, preventing them from interfering with each other's execution frequency.

Overall, these tests confirm that the rate_limit **decorator functions as intended, ensuring decorated functions are called within the specified rate limit and that each decorated function maintains its own independent rate limiting behavior.**

4. Testing Networking

```
import socket

import hypothesis.strategies as st
import pytest
from hypothesis import given
from port_scanner.networking import arp_scan, is_ip_address, is_port_open, ping,
tcp_syn_scan
from scapy.all import TCP  # type: ignore


@given(st.lists(st.integers(min_value=0, max_value=255), min_size=4, max_size=4))
# make a list of 4 numbers from 0-255
def test_is_ip_address_with_valid_addresses(numbers):
    address = ".".join(map(str, numbers))  # make ip address
    assert is_ip_address(address)
```

```
@given(
    st.lists(
        st.integers(min_value=-100000, max_value=-1) | st.integers(min_value=256,
max_value=100000),
        min_size=4,
        max_size=4,
    )
)
def test_is_ip_address_with_non_valid_octets(numbers):
    address = ".".join(map(str, numbers))  # make non valid ip address
    assert not is_ip_address(address)


@given(st.one_of(st.none(), st.booleans(), st.integers(), st.floats(),
st.lists(st.booleans())))
def test_is_ip_address_with_wrong_type(address):
    assert not is_ip_address(address)


def test_ping_localhost():
    assert ping("127.0.0.1")


def test_ping_invalid_ip_raises_value_error():
    with pytest.raises(ValueError):
        assert ping("255.256.0.0")


def test_is_port_open_localhost_closed_port(mocker):
    mock_socket = mocker.MagicMock(spec=socket.socket)
    mock_socket.connect.side_effect = TimeoutError(
        "Connection failed"
    )  # Simulate connection failure by raising an exception
    mocker.patch("socket.socket", return_value=mock_socket)
    assert not is_port_open("127.0.0.1", 80)


def test_is_port_open_localhost_open_port(mocker):
    mock_socket = mocker.MagicMock(spec=socket.socket)
    mock_socket.connect.return_value = None  # Simulate successful connection
    mocker.patch("socket.socket", return_value=mock_socket)
    assert is_port_open("127.0.0.1", 20)


def test_is_port_open_invalid_ip():
    with pytest.raises(ValueError):
        is_port_open("invalid", 20)


def test_is_port_open_invalid_port_type():
    with pytest.raises(TypeError):
        is_port_open("127.0.0.1", "20")  # type: ignore
```

```python
def test_is_port_open_port_not_in_range():
    with pytest.raises(ValueError):
        is_port_open("127.0.0.1", -1)
    with pytest.raises(ValueError):
        is_port_open("127.0.0.1", 100000)


def test_arp_scan(mocker):
    # Mock response packets
    mock_response = (
        [
            (mocker.MagicMock(), mocker.MagicMock(psrc="192.168.0.1")),
            (mocker.MagicMock(), mocker.MagicMock(psrc="192.168.0.2")),
        ],
        None,
    )

    # Set the return value of srp to the mocker response
    mocker.patch("port_scanner.networking.srp", return_value=mock_response)

    # Perform the ARP scan
    result = arp_scan("192.168.0.0/24")

    # Check if the function returns the expected list of IP addresses
    assert result == ["192.168.0.1", "192.168.0.2"]


def test_arp_scan_wrong_type():
    with pytest.raises(ValueError):
        arp_scan("invalid")


def test_tcp_syn_scan_open_port(mocker):
    # Mock a response with SYN-ACK flag set (indicating an open port)
    mock_response = mocker.MagicMock()
    mock_response.haslayer.return_value = True
    mock_response[TCP].flags = "SA"  # SYN-ACK flag

    # Set the return value of sr1 to the mocker response
    mocker.patch("port_scanner.networking.sr1", return_value=mock_response)

    # Perform the TCP SYN scan
    result = tcp_syn_scan("192.168.1.1", 80)

    # Check if the function returns True for an open port
    assert result


def test_tcp_syn_scan_closed_port(mocker):
    # Mock a response with RST flag set (indicating a closed port)
```

```
    mock_response = mocker.MagicMock()
    mock_response.haslayer.return_value = True
    mock_response[TCP].flags = "RA"  # RST flag

    # Set the return value of sr1 to the mocker response
    mocker.patch("port_scanner.networking.sr1", return_value=mock_response)

    # Perform the TCP SYN scan
    result = tcp_syn_scan("192.168.1.1", 80)

    # Check if the function returns False for a closed port
    assert not result


def test_tcp_syn_scan_filtered_port(mocker):
    # Mock a response with R flag set (indicating a filtered port)
    mock_response = mocker.MagicMock()
    mock_response.haslayer.return_value = True
    mock_response[TCP].flags = "R"  # Reset flag

    # Set the return value of sr1 to the mock response
    mocker.patch("port_scanner.networking.sr1", return_value=mock_response)

    # Perform the TCP SYN scan
    result = tcp_syn_scan("192.168.1.1", 80)

    # Check if the function returns False for a filtered port
    assert not result


def test_tcp_syn_scan_no_tcp(mocker):
    # Mock a response with R flag set (indicating a filtered port)
    mock_response = mocker.MagicMock()
    mock_response.haslayer.return_value = False

    # Set the return value of sr1 to the mock response
    mocker.patch("port_scanner.networking.sr1", return_value=mock_response)

    # Perform the TCP SYN scan
    result = tcp_syn_scan("192.168.1.1", 80)

    # Check if the function returns False for a filtered port
    assert not result


def test_tcp_syn_scan_other_status(mocker):
    # Mock a response with R flag set (indicating a filtered port)
    mock_response = mocker.MagicMock()
    mock_response.haslayer.return_value = True
    mock_response[TCP].flags = "X"

    # Set the return value of sr1 to the mock response
```

```
    mocker.patch("port_scanner.networking.sr1", return_value=mock_response)

    # Perform the TCP SYN scan
    result = tcp_syn_scan("192.168.1.1", 80)

    # Check if the function returns False for a filtered port
    assert not result


def test_tcp_syn_scan_no_response(mocker):
    # Set sr1 to return None (indicating no response)
    mocker.patch("port_scanner.networking.sr1", return_value=None)

    # Perform the TCP SYN scan
    result = tcp_syn_scan("192.168.1.1", 80)

    # Check if the function returns False for no response
    assert not result
```

This code defines a series of tests using the pytest framework to validate functionalities likely related to network scanning within the port_scanner.networking module. Here's a breakdown of what each test does and why it's important:

1. Testing is_ip_address **function:**

- **test_is_ip_address_with_valid_addresses:**
  - Uses hypothesis to generate valid IP addresses (4 numbers between 0-255 joined by dots).
  - Verifies the function correctly identifies valid IP addresses.
- **test_is_ip_address_with_non_valid_octets:**
  - Generates IP addresses with octets outside the valid range (0-255).
  - Ensures the function identifies these as invalid IP addresses.
- **test_is_ip_address_with_wrong_type:**
  - Tests various data types (None, booleans, integers, floats, lists of booleans) as input.
  - Confirms the function rejects non-string inputs for an IP address.
- These tests ensure the is_ip_address function accurately validates IP address formats, preventing unexpected behavior when working with network addresses.

2. Testing ping **function:**

- **test_ping_localhost:**
  - Pings the localhost address ("127.0.0.1") and asserts a successful response.
  - This verifies the basic functionality of the ping function for a known reachable host.
- **test_ping_invalid_ip_raises_value_error:**
  - Uses pytest.raises to expect a ValueError exception.
  - Attempts to ping an invalid IP address ("255.256.0.0").
  - This ensures the function handles invalid IP addresses appropriately.
- These tests confirm the ping function can identify reachable hosts and raise errors for invalid inputs.

3. Testing is_port_open **function:**

- **test_is_port_open_localhost_closed_port:**
    - Uses mocker to simulate a connection failure (TimeoutError) for a specific port on localhost (80).
    - Verifies the function identifies the port as closed.
- **test_is_port_open_localhost_open_port:**
    - Uses mocker to simulate a successful connection for a specific port on localhost (20).
    - Confirms the function identifies the port as open.
- **test_is_port_open_invalid_ip:**
    - Attempts to check a port with an invalid IP address ("invalid").
    - Ensures the function raises a ValueError for invalid IP inputs.
- **test_is_port_open_invalid_port_type:**
    - Tries to check a port with a non-integer port number (string "20").
    - Confirms the function raises a TypeError for invalid port number types.
- **test_is_port_open_port_not_in_range:**
    - Verifies the function raises a ValueError for ports outside the valid range (less than 1 or greater than 65535).

Why are these important?

- These tests ensure the is_port_open function can accurately determine open and closed ports, handle various input types, and reject ports outside the valid range.

4. Testing arp_scan **function:**

- **test_arp_scan:**
    - Uses mocker to simulate ARP scan responses for two IP addresses within a network range.
    - Verifies the function returns a list containing the discovered IP addresses.
- **test_arp_scan_wrong_type:**
    - Attempts to scan with an invalid input type (string "invalid") for the IP range.
    - Confirms the function raises a ValueError for invalid input types.

Why are these important?

- These tests validate the arp_scan function's ability to discover devices within a network range and its handling of invalid input formats.

5. Testing tcp_syn_scan **function:**

- **test_tcp_syn_scan_open_port:**
    - Uses mocker to simulate a response with a SYN-ACK flag (indicating an open port) for a specific IP and port.
    - Verifies the function identifies the port as open.

5. Conclusion

Through the execution of this extensive test suite, developers can acquire assurance regarding the dependability of the application. The tests cover a wide range of functionality, including managing errors and edge cases, network communication, and user input validation. This guarantees that the program operates as intended across various network conditions, completes scans with accuracy, and analyzes results meaningfully.