Software design report

Table of content

1.Introduction

This project consists of several Python files that work together to create a functional port scanner application. What does a functional port scanner do?

Port scanning is a method of determining which ports on a network are open and could be receiving or sending data. It is also a process for sending packets to specific ports on a host and analyzing responses to identify vulnerabilities.
This scanning can't take place without first identifying a list of active hosts and mapping those hosts to their IP addresses. This activity, called host discovery, starts by doing a network scan.
The goal behind port and network scanning is to identify the organization of IP addresses, hosts, and ports to properly determine open or vulnerable server locations and diagnose security levels. Both network and port scanning can reveal the presence of security measures in place such as a firewall between the server and the user's device.
After a thorough network scan is complete and a list of active hosts is compiled, port scanning can take place to identify open ports on a network that may enable unauthorized access.
It's important to note that network and port scanning can be used by both IT administrators and cybercriminals to verify or check the security policies of a network and identify vulnerabilities — and in the attackers' case, to exploit any potential weak entry points. In fact, the host discovery element in network scanning is often the first step used by attackers before they execute an attack.
As both scans continue to be used as key tools for attackers, the results of network and port scanning can provide important indications of network security levels for IT administrators trying to keep networks safe from attacks.

Here's an overview of each file we use to create this application and its purpose:

- **__main__.py**
  - Environment where top level code is run.
  - Imports all the other modules mentioned below.
  - This is the entry point of the application.
- **App.py:**
  - It utilizes the typer library to define command-line interface (CLI) commands for:

        - scan_arp: Performs an ARP scan of a provided IP range (utilizes functions from port_scanner.networking).
        - port_scan: Scans ports on a specified host within a port range (utilizes functions from port_scanner.networking).
    - It leverages the rich library for rich console output during the scan process. (a Python library for rich text and beautiful formatting in the terminal.)
    - It utilizes a logger from port_scanner.logger for recording messages during execution.
- **Decorators:**
    - This file contains decorator functions for the application.
    - Decorators are a way to modify the behavior of functions without permanently changing their code.
    - An example could be a rate_limit decorator to limit the rate of port scans to avoid overwhelming target systems.
- **Logger:**
    - This file defines a function get_logger that creates and configures a Python logging instance.
    - This allows for centralized logging of messages at different severity levels (debug, info, warning, error, critical) to a designated file.
- **Networking:**
    - This file contains core functionalities for network operations:
        - is_ip_address: Validates if a string is a valid IPv4 address.
        - ping (alternative: __ping using Scapy - commented out): Attempts to ping a host using either subprocess or the Scapy library.
        - is_port_open: Checks if a specific port on a host is open using socket connections.
        - arp_scan: Performs an ARP scan on a provided IP network using Scapy.
        - tcp_syn_scan: Implements a TCP SYN scan for a specific port on a host to determine if it's open or filtered.

2.Decorators

Architecture:
This code defines a single decorator function named rate_limit within the decorators.py file. The architecture follows a common pattern for decorators:

1. **Outer Decorator:** The rate_limit function acts as the outer decorator. It takes an argument (interval) specifying the rate limit in seconds.
2. **Inner Decorator:** The inner decorator function is responsible for the actual rate limiting logic. It's decorated with @wraps to preserve the original function's metadata.
3. **Wrapper Function:** Inside the decorator function, a wrapper function is defined. This wrapper function is what gets called when the decorated function is executed.

Purpose:
The primary purpose of this code is to implement a **rate limiting decorator**. This decorator can be applied to other functions to restrict their execution frequency. By specifying an interval, you can ensure that the decorated function is called at most once every interval

seconds. This helps to prevent overwhelming target systems during operations like port scanning or API calls.

Code explanation:

```
import time
from collections.abc import Callable
from functools import wraps
```

This line imports three libraries:

- time: Provides functions for measuring time and introducing delays, which might be useful for implementing rate limiting decorators.
- Callable from collections.abc: This imports the Callable type hint, which helps ensure the decorated function is indeed a callable object.
- wraps from functools: This imports the @wraps decorator, which is used to preserve the original function's metadata (like docstring and name) when decorating it.

```
def rate_limit(interval: float) -> Callable:

    def decorator(func: Callable):
        last_call_times = {}

        @wraps(func)
        def wrapper(*args, **kwargs):
            # Check if the function has been called before
            if func.__name__ not in last_call_times:
                last_call_times[func.__name__] = time.time()
            else:
                # Check if enough time has passed since the last call
                elapsed_time = time.time() - last_call_times[func.__name__]
                if elapsed_time < interval:
                    time.sleep(interval - elapsed_time)

            # Update the last call time
            last_call_times[func.__name__] = time.time()

            # Call the function and return the result
            return func(*args, **kwargs)

        return wrapper

    return decorator
```

This line defines a decorator function named rate_limit.

- interval: float: This defines a parameter named interval that expects a floating-point number representing the time interval (in seconds) for the rate limit.
- -> Callable: This type annotation indicates that the rate_limit function returns a callable object (the actual decorator).
- an inner function named decorator that will be used by the outer rate_limit function.
- last_call_times will be used to store the last call times for decorated functions.
- @wraps decorator from functools ensures that the wrapper function (defined below) inherits the docstring and name of the original function (func)
- *args, **kwargs – this makes sure that the wrapper function accepts any number of positional arguments and keyword arguments that are passed

- The block in the wrapper function checks if the decorated function has been called before and implements the rate limiting logic:
    - It checks if the function's name (func.__name__) exists as a key in the last_call_times dictionary.
    - If not, it means the function hasn't been called yet, so it adds the function name as a key and the current time (time.time()) as the value to the dictionary.
    - If the function has been called before, it calculates the elapsed time since the last call (elapsed_time).
    - If the elapsed time is less than the specified interval, it means the rate limit has been reached, so it uses time.sleep to introduce a delay equal to the difference (interval - elapsed_time) before proceeding.

- This line updates the last call time for the decorated function in the last_call_times dictionary with the current timestamp.

3.Logging

Architecture:

Function: It defines a single function named get_logger that takes a filename argument and returns a configured logger object.

Purpose:

- The primary purpose is to create and configure a logger instance for your program. This allows you to log messages at different severity levels throughout your code.

Code:

```
"""functions related to logging."""
import logging
def get_logger(filename: str) -> logging.Logger:
    """Get the program logger.
    Args:
    ____
        filename (str): The file to log to
    Returns:
    _______
```

```
        logging.Logger: a logger instance
    """
    logger = logging.getLogger(__name__)

    # the handler determines where the logs go: stdout/file
    file_handler = logging.FileHandler(filename)

    logger.setLevel(logging.DEBUG)
    file_handler.setLevel(logging.DEBUG)

    # the formatter determines what our logs will look like
    fmt_file = "%(levelname)s %(asctime)s [%(filename)s:%(funcName)s:%(lineno)d]
%(message)s"
    file_formatter = logging.Formatter(fmt_file)

    # here we hook everything together
    file_handler.setFormatter(file_formatter)
    logger.addHandler(file_handler)
    return logger
```

Key Functionalities:

1. **Logger Creation:**
    - logging.getLogger(__name__): Creates a logger object associated with the current module using its name (__name__).
2. **Handler Configuration:**
    - logging.FileHandler(filename): Creates a FileHandler object that will write logs to the specified filename. This determines where the logs are sent (a file in this case).
3. **Logging Level:**
    - logger.setLevel(logging.DEBUG) and file_handler.setLevel(logging.DEBUG): Sets the logging level to DEBUG for both the logger and the file handler. This defines the minimum severity level of messages that will be logged.
4. **Formatter Definition:**
    - logging.Formatter(fmt_file): Creates a Formatter object using a custom format string (fmt_file). This format specifies how log messages are formatted (including timestamp, level, filename, function name, line number, and message).
5. **Connecting Components:**
    - file_handler.setFormatter(file_formatter): Sets the defined formatter for the file handler, ensuring formatted messages are written to the file.
    - logger.addHandler(file_handler): Adds the file handler to the logger, enabling the logger to send messages to the handler for writing to the file.

4.Networking

Architecture:
The code consists of several independent functions, each serving a specific purpose:

- is_ip_address: Checks if a string represents a valid IPv4 address.
- ping: Utilizes either subprocess (system command) or a custom ICMP packet creation (__ping - marked for no coverage) to ping a host.
- is_port_open: Checks if a specific port on a host is open using a socket connection attempt.
- arp_scan: Performs an ARP scan on a provided IP network to discover devices.
- tcp_syn_scan: Sends a TCP SYN packet to a specific port on a host to determine if the port is open.

Purpose:

This file provides tools for network reconnaissance:

- Verifying if a host is reachable using ping.
- Checking if a specific port on a host is open for potential vulnerabilities.
- Discovering devices within a specific IP network range (ARP scan).

Relation with other files:

- This file likely interacts with other scripts in the project that require network information. For example, a port scanning script might call is_ip_address to validate user input and is_port_open to scan ports on discovered hosts.
- It might also be used by logging functionalities (like logger.py) to log network events.

```
import ipaddress
import platform
import re
import socket
import subprocess

from scapy.all import ARP, ICMP, IP, TCP, Ether, sr1, srp  # type: ignore

# regex that matches ipv4 addresses
IPV4_ADDRESS_PATTERN = re.compile(
    r""" # first octet
                    (25[0-5] # 250-255
                    |2[0-4][0-9] # 200-249
                    |[01]?[0-9][0-9]?) # 0-199
                    # other 3 octets
                    (\. # literal point
                    (25[0-5] # 250-255
                    |2[0-4][0-9] # 200-249
                    |[01]?[0-9][0-9]?)){3} # 0-199

""",
    re.VERBOSE,
)

MIN_PORT = 1  # lowest port that can be used
MAX_PORT = 65535  # highers port that can be used


def is_ip_address(address: str) -> bool:
    if not isinstance(address, str):
        return False
```

```
    return re.match(IPV4_ADDRESS_PATTERN, address) is not None


def ping(host: str) -> bool:
    if not is_ip_address(host):
        msg = "Host needs to be an ip address"
        raise ValueError(msg)
    # Option for the number of packets
    param = "-n" if platform.system().lower() == "windows" else "-c"

    # Building the command. Ex: "ping -c 1 google.com"
    command = ["ping", param, "1", host]

    return subprocess.call(command, stdout=subprocess.DEVNULL) == 0  # noqa: S603


def __ping(host: str) -> bool:  # pragma: no cover
    """Do not use, it's flaky"""
    if not is_ip_address(host):
        msg = "Host needs to be an ip address"
        raise ValueError(msg)
    # Craft an ICMP Echo Request packet (ping packet)
    icmp_packet = IP(dst=host) / ICMP(type=8)

    # Send the packet and receive a response
    response = sr1(icmp_packet, timeout=3, verbose=False)

    if response is not None:
        # Analyze the response
        if response.haslayer(ICMP):
            if response[ICMP].type == 0:  # ICMP Echo Reply
                return True
            elif response[ICMP].type == 3:  # ICMP Destination Unreachable  # noqa:
PLR2004
                return False
    return False


def is_port_open(host: str, port: int) -> bool:
    if not is_ip_address(host):
        msg = "host needs to be a valid ipv4 ip address"
        raise ValueError(msg)
    if not isinstance(port, int):
        msg = "port needs to be an integer"
        raise TypeError(msg)
    if port < MIN_PORT or port > MAX_PORT:
        msg = "port needs to be in between 1 and 65535"
        raise ValueError(msg)
    # creates a new socket
    s = socket.socket()
    try:
        s.settimeout(0.1)
```

```
        # tries to connect to host using that port
        s.connect((host, port))
    except TimeoutError:
        # cannot connect, port is closed
        # return false
        return False
    else:
        # the connection was established, port is open!
        return True


def arp_scan(ip_network: str) -> list[str]:
    try:
        ipaddress.ip_network(ip_network)
    except ValueError:
        msg = "Not a valid ip network"
        raise ValueError(msg) from None
    arp_request = ARP(pdst=ip_network)
    ether = Ether(dst="ff:ff:ff:ff:ff:ff")
    arp_broadcast = ether / arp_request
    answered_list, _ = srp(arp_broadcast, timeout=1, verbose=False)

    devices = []
    for _, received in answered_list:
        devices.append(received.psrc)

    return devices


def tcp_syn_scan(target_ip: str, target_port: int) -> bool:
    # Craft a TCP SYN packet
    syn_packet = IP(dst=target_ip) / TCP(dport=target_port, flags="S")

    # Send the packet and receive a response
    response = sr1(syn_packet, timeout=0.1, verbose=False)

    if response is not None:
        # Analyze the response
        if response.haslayer(TCP):
            if response[TCP].flags == "SA":
                return True
            elif response[TCP].flags == "RA":
                return False
            elif response[TCP].flags == "R":
                return False
    return False
```

Key Functionalities:

1. **IP Address Validation:**
    - is_ip_address ensures the validity of user-provided IP addresses used in other functions.

2. **Ping Functionality:**
    - ping offers two methods for pinging a host:
        - Utilizing subprocess.call to execute the system ping command (simpler but less flexible).
        - A commented-out __ping function that creates a custom ICMP Echo Request packet using the scapy library (more control over packet details but marked for no coverage).
3. **Port Scanning:**
    - is_port_open efficiently checks if a specific port on a host is open by attempting a socket connection.
4. **ARP Scanning:**
    - arp_scan utilizes the scapy library to perform an ARP scan on a provided IP network, discovering devices on that network.

5. **TCP SYN Scan:**
    - tcp_syn_scan (not covered in tests) demonstrates an advanced technique using scapy to send a TCP SYN packet and analyze the response to determine if a port is open.

Libraries Used:

- ipaddress: Provides functions for manipulating and validating IP addresses.
- platform: Helps determine the operating system for tailoring the ping command execution.
- re: Offers functionalities for regular expression matching (used for IP validation).
- socket: Enables creating sockets for network communication (used for port scanning).
- subprocess: Allows executing system commands (used for the basic ping implementation).
- scapy.all: A powerful library for crafting and manipulating network packets (used for custom ping and tcp_syn_scan).

5. Main Script (App.py)

Architecture:

The app.py file serves as the entry point for the port scanner application. It leverages the typer library to define a command-line interface (CLI) with two main commands: scan_arp and port_scan. These commands interact with functions from other modules (port_scanner.networking and port_scanner.decorators) to perform network operations like ARP scanning and port scanning.

Here's a line-by-line breakdown of app.py:

```
import ipaddress
import sys
from typing import Annotated

import typer
```

```
from rich.console import Console
from rich.live import Live
from rich.table import Table
```

These lines import necessary libraries:

- ipaddress: Used for IP address manipulation.
- sys: Provides access to system-specific parameters and functions (potentially used for error handling).
- typing: Used for type annotations (improving code readability and maintainability).
- typer: For building the CLI interface.
- rich: For rich console output features.

```
from port_scanner.decorators import rate_limit
```

- This line imports a rate_limit decorator function. This decorator is used to throttle port scans and avoid overwhelming target systems.

```
from port_scanner.logger import get_logger
```

- Imports the get_logger function from port_scanner.logger. This function creates and configures a logger instance for recording messages during application execution.

```
from port_scanner.networking import is_ip_address, is_port_open, ping,
tcp_syn_scan
```

- Imports the functions is_ip_address, is_port_open, ping, and tcp_syn_scan from the port_scanner.networking file, which we'll talk about later in this software design report.

```
LOGGER = get_logger("port-scan.log")
```

- Creates a logger instance named LOGGER using the get_logger function. It will log messages to a file named "port-scan.log".

```
app = typer.Typer(no_args_is_help=True)
```

- Initializes a typer application instance named app.
  The no_args_is_help=True argument ensures that running the script without arguments displays the help message.

```
console = Console()
```

- Creates a Console instance from the rich library for interacting with the console and displaying rich output.

```
def _typer_check_host(host: str) -> str:
    if not is_ip_address(host):
        msg = "Host needs to be an ip address"
        raise typer.BadParameter(msg)
    else:
         return host
```

- Defines a custom function _typer_check_host that utilizes the is_ip_address function (from port_scanner.networking) to validate user input for the host (ensuring it's a valid IP address) when using the CLI commands. If the validation fails, it raises a typer.BadParameter exception with an informative message.

```
def _typer_check_range(ip_range: str):
        ipaddress.IPv4Network(ip_range)
        return ip_range
    except (ipaddress.AddressValueError, ipaddress.NetmaskValueError):
        msg = "Host needs to be a valid ip range (ip/mask)"
        raise typer.BadParameter(msg) from None
```

Defines a custom function _typer_check_range that utilizes the ipaddress library to validate user input for the IP range when using the scan_arp command. It ensures the provided string is a valid IPv4 network format. If the validation fails, it raises a typer.BadParameter exception with an informative message.

```
@app.command()
def scan_arp(ip_range: Annotated[str,
typer.Option(callback=_typer_check_range, prompt=True)]):
    """perform an arp scan of the ip-range."""
    from port_scanner import networking

    devices = networking.arp_scan(ip_range)
    table = Table()
    table.add_column("device ip address")
    for device in devices:
        table.add_row(device)
    console.print(table)
```

Define scan_arp **command:**

- Takes an optional ip_range argument (validated with _typer_check_range).
- Docstring: Briefly describes the purpose.
- Provides access to the arp_scan function.
- Call networking.arp_scan(ip_range) to scan the network.
- Store discovered devices in devices.
- Build a table to display results.
- Add a column named "device ip address".

- Iterate through discovered devices (devices).
- Add each device's IP address as a new row.
- Use console.print(table) to display the results.

```
@app.command()
def port_scan(
    host: Annotated[str, typer.Option(callback=_typer_check_host,
prompt=True)],
    start_port: Annotated[int, typer.Option(prompt=True)],
    end_port: Annotated[int, typer.Option(prompt=True)],
    wait_between_ports: Annotated[float, typer.Option()] = 0,
    use_tcp_syn: bool = False,  # noqa: FBT001, FBT002
    skip_ping: bool = False,  # noqa: FBT002, FBT001
) -> None:
    """Scan host's ports from start-port to end-port"""
    if not skip_ping:
        if ping(host):
            console.print(f"{host} seems to be up")
            LOGGER.info(f"{host} seems to be up")
        else:
            console.print(f"{host} could not be pinged")
            LOGGER.error(f"{host} could not be pinged")
            sys.exit(1)

    if use_tcp_syn:
        scan = tcp_syn_scan
    else:
        scan = is_port_open
    if wait_between_ports:
        scan = rate_limit(wait_between_ports)(scan)
    table = Table()
    table.add_column("Port")
    table.add_column("Status")
    with Live(table, refresh_per_second=4):
        ports = range(max(1, start_port), min(65535, end_port + 1))
        for port in ports:
            response = scan(host, port)
            if response:
                LOGGER.info(f"port {port} on {host} is open")
                table.add_row(f"{port}", "[green]open[/]")
            else:
                LOGGER.info(f"port {port} on {host} is closed")
                table.add_row(f"{port}", "[red]closed[/]")
```

User Input and Validation:

- Takes arguments for host (validated with _typer_check_host), start_port, end_port (all prompted from the user).
- wait_between_ports is optional (default 0).
- use_tcp_syn and skip_ping are optional flags (default False).

Ping Check:

- If skip_ping is False, it attempts to ping the host and logs/prints success or failure. Exits on ping failure (configurable).

Scan Function Selection:

- Chooses tcp_syn_scan for TCP SYN scan if use_tcp_syn is True, otherwise defaults to is_port_open.
- Optionally applies a rate limiter (rate_limit) if wait_between_ports is set.

Port Scan Loop:

- Creates a table for displaying results.
- Iterates through a validated port range (ensuring valid values).
- Calls the chosen scan function (scan) for each port and host combination.
- Logs scan results and updates the table with port status (open/closed) in green/red colors.

Live Table Update:

- Uses rich library's Live functionality to refresh the table every 4 seconds, providing a dynamic view of the scan progress.

6.Justification for Libraries

Here's a breakdown of the libraries encountered so far and why they are justified choices:

Common Libraries:

- ipaddress: This library is essential for working with IP addresses in Python. It provides functions for validating, manipulating, and converting IP addresses in various formats (v4 and v6). These functionalities are crucial for functions like is_ip_address, ping, arp_scan, and potentially others that interact with IP addresses.
- platform: This library is helpful for adapting code execution based on the operating system. In networking.py, it's used with the ping function to determine the appropriate command flag (-c for Linux/macOS, -n for Windows) for consistent ping behavior across different platforms.
- re: The re (regular expression) library is a versatile tool for pattern matching and string manipulation. In networking.py, it's used within the is_ip_address function to define a regular expression that validates the format of a provided string as a valid IPv4 address.

decorators.py**:**

- functools: This library provides the @wraps decorator, which is used in rate_limit to preserve the original function's metadata (docstring and name) when decorating it. This ensures decorated functions retain their proper documentation and identification within the codebase.
- time: The time library offers functionalities for measuring time and introducing delays. In the rate_limit decorator, it's used to calculate elapsed time since the last function call and potentially introduce a delay to enforce the rate limit.

logger.py**:**

- logging: This is the core library for implementing logging functionalities in Python applications. It provides tools for creating loggers, defining logging levels, configuring handlers (like file handlers), and formatting log messages. logger.py utilizes these functionalities to create a well-configured logger object for writing logs to a file.

networking.py

- socket: The socket library is fundamental for network communication in Python. The is_port_open function directly uses sockets to create a connection attempt to a specific port on a host, ultimately determining if the port is open.
- subprocess**:** This library allows executing system commands from within Python code. In networking.py, the basic ping implementation utilizes subprocess.call to run the system's ping command for host reachability checks. While convenient, it offers less control over the ping process compared to the commented-out __ping function using scapy.
- scapy.all **(Optional):** The scapy library is a powerful tool for crafting, sending, and analyzing network packets. It provides a rich set of functionalities for various network protocols. In networking.py, the commented-out __ping function demonstrates creating a custom ICMP Echo Request packet (ping packet) for a more controlled ping approach. Similarly, the tcp_syn_scan function (marked for no coverage) utilizes scapy to send a TCP SYN packet and analyze the response for advanced port scanning techniques. While scapy offers more control, it also introduces an additional dependency compared to the basic functionalities.

Overall Justification:

The choice of libraries aligns well with the functionalities provided in each file. Libraries like ipaddress, platform, re, functools, time, logging, socket, and optionally scapy are justified due to their specific functionalities that directly contribute to the core functionalities of the code. They provide efficient ways to manipulate IP addresses, adapt to different platforms, validate user input, enforce rate limits, manage logging, create network connections, and optionally craft and analyze network packets, ultimately enabling the network-related tasks within the application.