Nmap

What is nmap?

Nmap (Network Mapper) is an open-source tool used for network discovery and security auditing. It is primarily used to scan networks, identify devices, and detect open ports and services.

I have developed a simplified version of Nmap using Python, which is capable of scanning TCP, UDP, and half-TCP ports. This project allows for basic network discovery and service detection, similar to the functionality of Nmap but on a smaller scale, enabling users to analyze open ports and services running on target systems.

Introduction

This report summarizes the development and results of a project where a lightweight version of Nmap was created using Python. The tool was designed to scan TCP, UDP, and half-TCP ports on a target network to discover open ports and identify active services. The primary objective of the project was to replicate key functionalities of Nmap, while focusing on efficiency and simplicity, making the tool suitable for network administrators and security professionals looking for a basic yet effective solution for network scanning.

Project Description and Methodology

The project aimed to build a network scanning tool capable of scanning different types of ports: TCP, UDP, and half-TCP (a partial TCP handshake). The approach involved the following key steps:

1. **Network Scanning**:
   - Utilized Python's socket library to implement the scanning of ports on a target IP address.
   - Supported both full TCP scans and UDP scans, with half-TCP scanning implemented using a modified TCP connection attempt.
2. **Port Scanning**:
   - Implemented functions to scan a range of ports for each protocol type.
   - Used non-blocking I/O for efficient scanning, ensuring that timeouts and delays were managed appropriately.
3. **User Interface**:
   - The tool was designed to accept user input for specifying the target IP and the range of ports to scan.
   - Simple textual output was provided to indicate the status of each scanned port.
4. **Testing**:
   - Conducted several test scans on local networks and isolated systems to ensure accuracy in detecting open ports and services.

Results/Findings

The Python-based tool successfully scanned both TCP and UDP ports, providing the user with a clear report of open ports. The half-TCP scanning feature simulated the first part of a TCP handshake, effectively detecting ports that respond to partial connection attempts.

Key findings include:

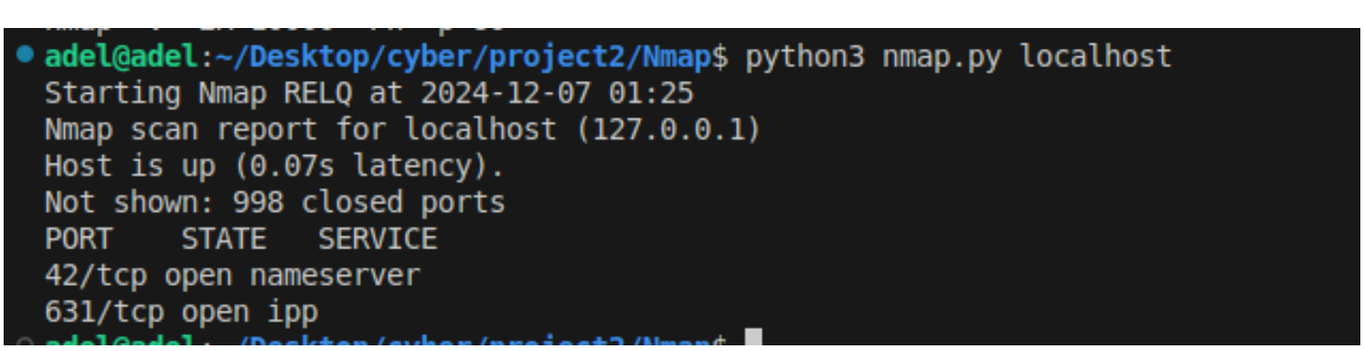
- **Port Detection**: The tool reliably detected open TCP and UDP ports on the scanned target systems.
- **Half-TCP Scanning**: Half-TCP scanning provided a way to identify services that respond to incomplete connection attempts, mimicking a stealthy scan approach.
- **Performance**: The tool performed adequately on small networks, though larger networks required optimizations in threading and timeout handling to improve scan time.

Lessons Learned

1. **Timeout Management**: Network timeouts and delays significantly impacted scan results, especially during UDP scans. Implementing non-blocking I/O and timeouts helped mitigate this.
2. **Efficiency with Multi-threading**: Scanning large networks was slow without multi-threading. The use of threading helped improve the overall scan speed.
3. **Port Scanning Accuracy**: While TCP and UDP scans were straightforward, half-TCP scanning proved to be a more complex implementation, requiring deeper understanding of TCP handshake mechanics.
4. **Error Handling**: Robust error handling for network errors and unreachable targets was essential to improve the tool's reliability.

How to use

- **Default TCP scan for first 1000 ports**



- **-sU for UDP scan**

```
adel@adel:~/Desktop/cyber/project2/Nmap$ python3 nmap.py localhost -sU -p42
Starting Nmap RELQ at 2024-12-07 01:30
Nmap scan report for localhost (127.0.0.1)
Host is up (0.10s latency).
PORT    STATE   SERVICE
42/udp  closed  nameserver
```

- **-p <port> scans for the port**

```
adel@adel:~/Desktop/cyber/project2/Nmap$ python3 nmap.py localhost -p13
Starting Nmap RELQ at 2024-12-07 01:29
Nmap scan report for localhost (127.0.0.1)
Host is up (0.08s latency).
PORT    STATE   SERVICE
13/tcp  closed  daytime
```

- **-p <port-port2> scans for the range of port**

```
adel@adel:~/Desktop/cyber/project2/Nmap$ python3 nmap.py localhost -p 20-30
Starting Nmap RELQ at 2024-12-07 01:30
Nmap scan report for localhost (127.0.0.1)
Host is up (0.07s latency).
PORT    STATE   SERVICE
20/tcp closed ftp-data
21/tcp closed ftp
22/tcp closed ssh
23/tcp closed telnet
24/tcp closed priv-mail
25/tcp closed smtp
26/tcp closed rsftp
27/tcp closed nsw-fe
28/tcp closed unknown
29/tcp closed msg-icp
30/tcp closed unknown
```

Implementation Steps

Understanding Sockets The project began with studying sockets, a fundamental concept for network communication. A socket provides an endpoint for sending and receiving data across a network. I focused on two primary types of sockets:

- **TCP Sockets**: These ensure reliable communication and are connection-oriented.
- **UDP Sockets**: These are connectionless and prioritize speed over reliability.

By understanding the differences between these protocols, I could determine how to use them for different types of port scans.

Creating a TCP Socket

- The first step was creating a TCP socket using Python's socket module.
- I used the socket.AF_INET and socket.SOCK_STREAM constants to initialize the socket.
- To test port availability, the script attempted to establish a connection to a target host and port. A successful connection indicated that the port was open.

Creating a UDP Socket

- Next, I implemented UDP scanning, which was more challenging because UDP does not provide acknowledgment for successful communication.
- I created a UDP socket using socket.SOCK_DGRAM and sent a datagram to the target host and port. If a response was received or an error returned, I inferred the port's status.

```
############################################################
def socket_setup(type):
    if type == "udp":
        return socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    elif type == "tcp":
        return socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    elif type == "syn":
        return socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_TCP)

############################################################
```

- I used a timeout to handle cases where no response was received, avoiding indefinite blocking and improving scan speed.

```
def check_port(ip, port, server, type):
    server.settimeout(3)
    if type == "udp":
        try:
            server.sendto(b"", (ip, port))
            server.recvfrom(1024)
            if socket.timeout:
                return 111
            return 0
        except KeyboardInterrupt:
            exit(1)
        except socket.error as e:
            if e.errno == 111:
                return 111
            return 111
```

The socket.connect_ex() method in Python is a non-blocking version of the socket.connect() method. It attempts to connect to the specified address but, instead of raising an exception on error, it returns an error code.

Error Code	Meaning	Platform
`0`	Connection successful	All
`111`	Connection refused	Linux/Unix
`110`	Connection timed out	Linux/Unix
`113`	No route to host	Linux/Unix
`10061`	Connection refused	Windows
`10060`	Connection timed out	Windows

- 0 for open
- 111 for closed
- 110 for filtered

Implementing Half-TCP Scans

- A half-TCP (SYN) scan was implemented by sending a SYN packet and analyzing the response.

- I crafted the SYN packets manually by constructing both the TCP and IP headers myself, ensuring full control over the data being sent.
- This required a deep understanding of the structure of TCP and IP headers and involved setting specific flags and fields.
- To achieve this, I utilized raw sockets (socket.SOCK_RAW), which provided the ability to send custom-crafted packets directly to the network layer without interference from the operating system's protocol stack.

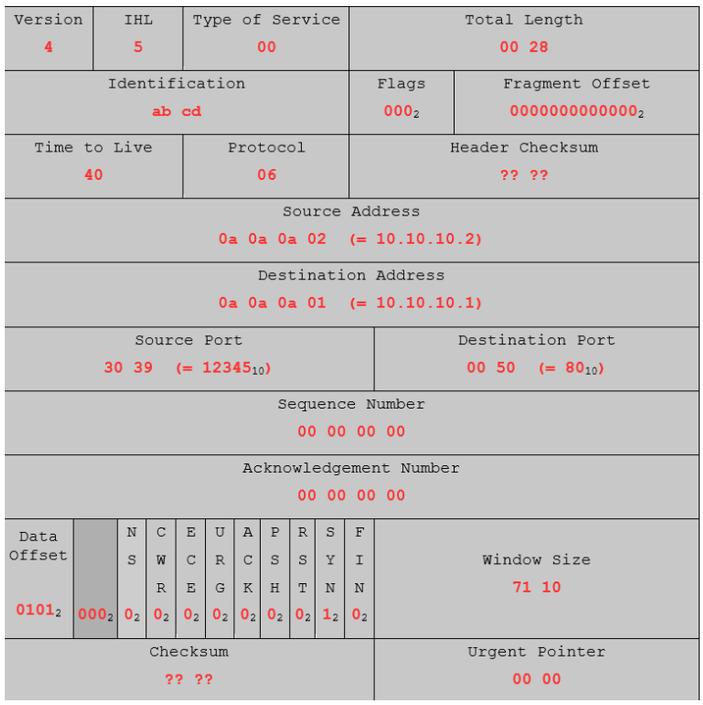
Creating a Raw Socket

- Raw sockets allow for the customization of packet headers, enabling advanced features like SYN scanning.
- Using raw sockets requires root privileges because of the potential to interfere with network communications.
- I used Python's struct module to construct headers in the required binary format.

```
create_ip_header(source_ip, dest_ip):
ip_ver = 4
ihl = 5
version_ihl = (ip_ver << 4) + ihl
tos = 0
total_length = 40
packet_id = 54321
fragment_offset = 0
ttl = 64
protocol = socket.IPPROTO_TCP
checksum_placeholder = 0
source_ip = socket.inet_aton(source_ip)
dest_ip = socket.inet_aton(dest_ip)

ip_header = struct.pack('!BBHHHBBH4s4s',
                        version_ihl,
                        tos,
                        total_length,
                        packet_id,
                        fragment_offset,
                        ttl,
                        protocol,
                        checksum_placeholder,
                        source_ip,
                        dest_ip)
ip_checksum = checksum(ip_header)
ip_header = struct.pack('!BBHHHBBH4s4s',
                        version_ihl,
                        tos,
                        total_length,
                        packet_id,
                        fragment_offset,
                        ttl,
                        protocol,
                        ip_checksum,
                        source_ip,
                        dest_ip)
return ip_header
```

```
f create_tcp_header(source_ip, dest_ip, source_port, dest_port):
 seq = 0
 ack_seq = 0
 data_offset = (5 << 4)
 flags = 0x02  # SYN flag
 window = socket.htons(5840)
 checksum_placeholder = 0
 urgent_pointer = 0

 tcp_header = struct.pack('!HHLLBBHHH',
                          source_port,
                          dest_port,
                          seq,
                          ack_seq,
                          data_offset | flags,
                          0,
                          window,
                          checksum_placeholder,
                          urgent_pointer)

 pseudo_header = struct.pack('!4s4sBBH',
                             socket.inet_aton(source_ip),
                             socket.inet_aton(dest_ip),
                             0,
                             socket.IPPROTO_TCP,
                             len(tcp_header))

 pseudo_packet = pseudo_header + tcp_header
 tcp_checksum = checksum(pseudo_packet)

 tcp_header = struct.pack('!HHLLBBHHH',
                          source_port,
                          dest_port,
                          seq,
                          ack_seq,
                          data_offset | flags,
                          0,
                          window,
                          tcp_checksum,
                          urgent_pointer)
 return tcp_header
```

This is an understandable illustration of how the headers should be structured in a packet.



Conclusion

The project successfully developed a basic network scanning tool that replicates core functionalities of Nmap using Python. It can scan TCP, UDP, and half-TCP ports on target systems, providing essential network discovery and service detection capabilities. The tool demonstrated its usefulness for small-scale network analysis, although scalability improvements would be necessary for larger networks.