Creating a SYN port scanner

In this chapter we are going to use our knowledge on packets to create our own SYN port scanner.

A very simple port scanner

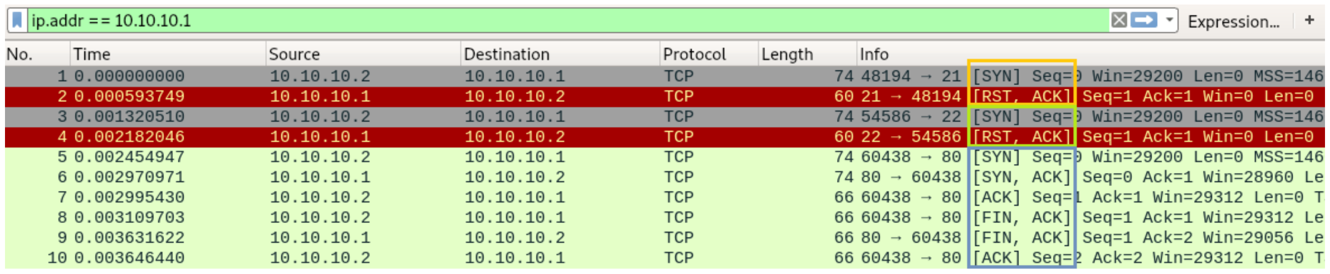
Port scanners are tools designed to probe a server for open ports. They are used by many people such as administrators or pentesters to check the attack surface of their systems and identify networks services running on them. One of the most well-known port scanners is nmap. These port scanners typically allow several different kind of scanning techniques. The most basic one is the TCP connect scan. With this technique the port scanner tries to complete a full TCP three-way handshake (`[SYN], [SYN ACK], [ACK]`, see Recap on network layers and protocols) with the targeted port on the system to scan. The following Python script does exactly this:

```python
import socket

def scan(host, port):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        s.connect((host, port))
        print("Port open: " + str(port))
        s.close()
    except:
        print("Port closed: " + str(port))

host = "10.10.10.1"
for port in [21, 22, 80]:
    scan(host, port)
```

```
root@kali:~# python3 simple_scanner.py
```

```
Port closed: 21
Port closed: 22
Port open: 80
```

In Wireshark we can observe the exact behavior of our simple port scanner:



ip.addr == 10.10.10.1

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.10.10.2	10.10.10.1	TCP	74	48194 → 21 [SYN] Seq=0 Win=29200 Len=0 MSS=146
2	0.000593749	10.10.10.1	10.10.10.2	TCP	60	21 → 48194 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
3	0.001320510	10.10.10.2	10.10.10.1	TCP	74	54586 → 22 [SYN] Seq=0 Win=29200 Len=0 MSS=146
4	0.002182046	10.10.10.1	10.10.10.2	TCP	60	22 → 54586 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
5	0.002454947	10.10.10.2	10.10.10.1	TCP	74	60438 → 80 [SYN] Seq=0 Win=29200 Len=0 MSS=146
6	0.002970971	10.10.10.1	10.10.10.2	TCP	74	80 → 60438 [SYN, ACK] Seq=0 Ack=1 Win=28960 Le
7	0.002995430	10.10.10.2	10.10.10.1	TCP	66	60438 → 80 [ACK] Seq=1 Ack=1 Win=29312 Len=0 T
8	0.003109703	10.10.10.2	10.10.10.1	TCP	66	60438 → 80 [FIN, ACK] Seq=1 Ack=1 Win=29312 Le
9	0.003631622	10.10.10.1	10.10.10.2	TCP	66	80 → 60438 [FIN, ACK] Seq=1 Ack=2 Win=29056 Le
10	0.003646440	10.10.10.2	10.10.10.1	TCP	66	60438 → 80 [ACK] Seq=2 Ack=2 Win=29312 Len=0 T

We are scanning for the three ports 21, 22 and 80. For the first two ports we see that they are closed as we receive a `[RST ACK]` packet for each of them from the server. However, port 80 is open as we completed a TCP three-way handshake as well as a normal TCP connection termination initiated by us after that.

This kind of port scan is even possible if the user does not have the privileges to create raw packets on the system as the script asks the underlying operating system to establish the connection with the target machine by a `connect` system call. However, this method is quite noisy, and the target machine is very likely to log the connection.

TCP SYN Scan

A faster and a little bit more stealthy port scan is a SYN scan. This is probably the most common technique for port scanners in general. Often also referred to as half-open scanning, because you don't do a full TCP connection. You send a `[SYN]` packet and then wait for the response. A `[SYN ACK]` indicates that the port is open, while a `[RST]` indicates that it's closed. In contrast to the connect scan, in this method of scanning we do not complete the three-way handshake by sending a `[ACK]` ourselves, but terminate the connection.

Looking back at an excerpt from the script with which we created a TCP/IP packet, what values would we need to change for a port scanner use case?

```
ip_header  = b'\x45\x00\x00\x28' # Version, IHL, Type of Service | Total
ip_header += b'\xab\xcd\x00\x00' # Identification | Flags, Fragment Offs
ip_header += b'\x40\x06\xa6\xec' # TTL, Protocol | Header Checksum
ip_header += b'\x0a\x0a\x0a\x02' # Source Address
ip_header += b'\x0a\x0a\x0a\x01' # Destination Address

tcp_header  = b'\x30\x39\x00\x50' # Source Port | Destination Port
tcp_header += b'\x00\x00\x00\x00' # Sequence Number
tcp_header += b'\x00\x00\x00\x00' # Acknowledgement Number
tcp_header += b'\x50\x02\x71\x10' # Data Offset, Reserved, Flags | Windc
tcp_header += b'\xe6\x32\x00\x00' # Checksum | Urgent Pointer
```

The most obvious changes necessary are probably the Destination Address and Destination Port. Changing those parameters makes it necessary for us to also adjust the Header Checksum and the Checksum from the TCP segment on the fly.

Let’s put together what our script needs to do:

1. Take our target IP address and port as input
2. Insert them into a blueprint for a `[SYN]` packet
3. Calculate the new checksums
4. Send the packet
5. Check the response for `[SYN, ACK]`, if present: port open, if not: port closed
6. Repeat all steps above (with a different port)

Congratulations, you are now able to create your own TCP SYN port scanner! In order to be a little bit more flexible, a rewrite of the previous scripts should be done. The struct Python module helps in working with the binary data. In case you want to see how the steps above could look like in code, have a look at the following lines.

```
import socket
from struct import *
import binascii


class Packet:
    def __init__(self, src_ip, dest_ip, dest_port):
```

```
# https://docs.python.org/3.7/library/struct.html#format-charact
# all values need to be at least one byte long (-> we need to ac

############
# IP segment
self.version = 0x4
self.ihl = 0x5
self.type_of_service = 0x0
self.total_length = 0x28
self.identification = 0xabcd
self.flags = 0x0
self.fragment_offset = 0x0
self.ttl = 0x40
self.protocol = 0x6
self.header_checksum = 0x0
self.src_ip = src_ip
self.dest_ip = dest_ip
self.src_addr = socket.inet_aton(src_ip)
self.dest_addr = socket.inet_aton(dest_ip)
self.v_ihl = (self.version << 4) + self.ihl
self.f_fo = (self.flags << 13) + self.fragment_offset

#############
# TCP segment
self.src_port = 0x3039
self.dest_port = dest_port
self.seq_no = 0x0
self.ack_no = 0x0
self.data_offset = 0x5
self.reserved = 0x0
self.ns, self.cwr, self.ece, self.urg, self.ack, self.psh, self.
self.window_size = 0x7110
self.checksum = 0x0
self.urg_pointer = 0x0
self.data_offset_res_flags = (self.data_offset << 12) + (self.re

########
# packet
self.tcp_header = b""
self.ip_header = b""
self.packet = b""
```

```
    def calc_checksum(self, msg):
        s = 0
        for i in range(0, len(msg), 2):
            w = (msg[i] << 8) + msg[i+1]
            s = s + w
        # s = 0x119cc
        s = (s >> 16) + (s & 0xffff)
        # s = 0x19cd
        s = ~s & 0xffff
        # s = 0xe632
        return s


    def generate_tmp_ip_header(self):
        tmp_ip_header = pack("!BBHHHBBH4s4s", self.v_ihl, self.type_of_s
                                              self.identification, self.f_fo
                                              self.ttl, self.protocol, self.
                                              self.src_addr,
                                              self.dest_addr)
        return tmp_ip_header


    def generate_tmp_tcp_header(self):
        tmp_tcp_header = pack("!HHLLHHHH", self.src_port, self.dest_port
                                           self.seq_no,
                                           self.ack_no,
                                           self.data_offset_res_flags, self.
                                           self.checksum, self.urg_pointer)
        return tmp_tcp_header


    def generate_packet(self):
        # IP header + checksum
        final_ip_header = pack("!BBHHHBBH4s4s", self.v_ihl, self.type_of
                                                self.identification, sel
                                                self.ttl, self.protocol,
                                                self.src_addr,
                                                self.dest_addr)
        # TCP header + checksum
        tmp_tcp_header = self.generate_tmp_tcp_header()
        pseudo_header = pack("!4s4sBBH", self.src_addr, self.dest_addr,
        psh = pseudo_header + tmp_tcp_header
        final_tcp_header = pack("!HHLLHHHH", self.src_port, self.dest_pc
                                             self.seq_no,
                                             self.ack_no,
                                             self.data_offset_res_flags,
                                             self.calc_checksum(psh), se

        self.ip_header = final_ip_header
        self.tcp_header = final_tcp_header
        self.packet = final_ip_header + final_tcp_header


    def send_packet(self):
        s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROT
        s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)
        s.sendto(self.packet, (self.dest_ip, 0))
        data = s.recv(1024)
        s.close()
        return data



# could work with e.g. struct.unpack() here
# however, lazy PoC (012 = [SYN ACK]), therefore:
def check_if_open(port, response):
    cont = binascii.hexlify(response)
    if cont[65:68] == b"012":
        print("Port "+str(port)+" is: open")
    else:
        print("Port "+str(port)+" is: closed")




for port in [21, 22, 80, 8080]:
    p = Packet("10.10.10.2", "10.10.10.1", port)
    p.generate_packet()
    result = p.send_packet()
    check_if_open(port, result)
```

Small exercise: Compare the difference in the packets you get back from your own local test server (10.10.10.1), compared to for example the IP address of google.com on port 21 and port 80. How could you improve the upper script?

Final words

As you have seen, it's very much possible to create and send raw packets manually, even in high level languages such as Python. However, a more comfortable way of working with sockets, packets and networking in general would be Scapy, an interactive packet manipulation program for Python. If you are interested in that area, I highly recommend you have a look at it. Overall, I hope you did learn a few things from this series, and maybe can use that knowledge for example in your next fuzzing project, pentest or as a basis for further improvement.

A word of warning: Use port scanners only on your own systems or servers where you have the explicit permission to scan. Depending of the jurisdiction you and/or your target system are in, you might get in conflict with the law when performing unsolicited port scans.

TCP/IP packets

Contact

Twitter: @inc0x0