Portscanner

Mirko Bez Simon Targa

January 19, 2016

Contents

Introduction

The first goal of the project was to understand the theory behind port scanners and their detection. The next goal was to implement a port scanner. The scanner should be written in C and support various scan methods (e.g. TCP connect scan, TCP SYN scan ...). The last goal was to implement our own port scan detector.

The final result consists of two programs: a port scanner and a port scan detector. The port scanner tries to simulate the behavior of nmap, which is one of the most used programs for port scanning. The port scan detector uses the pcap library in order to sniff the incoming network packets to recognize port scan attempts.

This document describes how the two programs work and the theory behind them. The focus of section 1.1 is on the TCP connect scan which is the most simple port scan technique. Section 1.2 is about how the TCP SYN scan works and how it was implemented within the scope of this project. Section 1.3 describes the scan methods Xmas, TCP NULL and Fin scan and their implementations. Section 2 is dedicated on how port scanning attempts can be detected and blocked by an IT administrator.

Motivation

Port scanners are used to determine which ports are open. This information can be used by attackers to identify services running on a host and exploit vulnerabilities.

For example, researchers recently identified bugs in Oracle's Java SE that allow arbitrary execution of code, access to security sensitive data, unauthorized changes in security configurations, and so on [**?**].

1 Port Scannner

1.1 TCP connect scan

1.1.1 Theory

The TCP connect scan is probably the most easy method to scan for open ports. It simply takes advantage of the system call *connect* of the underlying operating system, in order to establish a connection with the target machine and port. Afterwards the returned value of the system call is used to determine if the port to check is either closed or open at the target machine [**?**].

1.1.2 Details of implementation

In order to use the system call `connect` the implementation uses C sockets of the type `SOCK_STREAM`. This type of socket allows us, to establish a tcp connection to the target machine.

```
int mysocket;
mysocket=socket(AF_INET, SOCK_STREAM, 0);
```

Listing 1.1: C code to create a tcp socket in C

Additionally to the socket we also have to use a structure of the type `sockaddr_in` to connect to the target machine and port. The structure is needed to define the ip address of the target machine and the port to use for the connection. The listing shows the code of how to assign the ip address and port to a structure of the type `sockaddr_in`.

```
struct sockaddr_in server;
struct hostent *host;
hostname = gethostbyname(p->host_name);
memcpy( (char *)&server.sin_addr, host->h_addr_list[0], host->h_length);
server.sin_family = AF_INET;
server.sin_port = htons(port);
```

Listing 1.2: C code to use the structure `sockaddr_in`

The last step is to use the created socket and `sockaddr_in` structure to connect to the target machine and port. If the connection could be established we know that the port is open. To check if the connection attempt was successful, we only have to check the return value of the connect() function. Upon successful completion, connect() shall return 0. The code to use the connect() function is shown in the listing.

```
if(connect(mysocket, (struct sockaddr *)&server, sizeof(server))>=0){
  printf("TCP␣-␣Port␣%d␣is␣open\n", i);
  close(mysocket);
  mysocket = socket(AF_INET, SOCK_STREAM, 0);
}
```

Listing 1.3: C code to use the connect() to check if port is open

1.2 TCP SYN scan

1.2.1 Theory

1.2.2 Details of implementation

In order to only send a syn request instead of open a full tcp connection (including handshake) the implementation uses raw sockets. Raw sockets allow to control every section of the packets that will be sent. The function socket(), as shown in listing, can be used to create a raw socket that uses the tcp protocol.

```
int mysocket;
mysocket=socket(AF_INET, SOCK_RAW,  IPPROTO_TCP);
```

Listing 1.4: C code to use the connect() to check if port is open

Before we can send a syn request to the target machine, we have to build the packet to be sent. To send packets with a raw socket the function sendto() is used. It's second parameter is a pointer to the message to be sent, which is the packet that we build. It consists of the tcp theader, the ip header and the data to be sent. As we only want to send a syn request we don't care about the data, therefore it is empty. We start building the packet by filling in the IP-Header. We don't need optional fields therefore the we use the minimal size possible size of the ip header which is 160 Bits (5*32 Bits). We use the ip version 4, which is still the most widely used ip version. The length of our packet is the sum of the length of the ip header and the length of the tcp header. For the time to live we choose 64, which should be big enough fur our purpose. As transfer protocol we set the tcp protocol. The source address of the ip header is set to the ip address of the scanning system and the destination address is set to the address of the target system to scan. To have a complete ip header we also have to calculate its check sum.

```
//Fill in the IP Header
iph->ihl = 5;
iph->version = 4;
iph->tos = 0;
iph->tot_len = sizeof (struct ip) + sizeof (struct tcphdr);
iph->id = htons (54321); //Id of this packet
iph->frag_off = htons(16384);
iph->ttl = 64;
iph->protocol = IPPROTO_TCP;
iph->saddr = inet_addr ( source_ip );
iph->daddr = dest_ip.s_addr;
iph->check = csum(datagram, iph->tot_len>>1);
```

Listing 1.5: C code to fill in ip header

Before the packet can be sent, we also need to fill in the tcp header. In order to send a syn request we only set the syn flag to true and all the other flags to false.

```
tcph->fin=0;
tcph->syn=1;
tcph->rst=0;
tcph->psh=0;
tcph->ack=0;
tcph->urg=0;
```

Listing 1.6: C code to set flags in tcp header

The last step before we can send the packet is to set the destination port (the port to scan) in the tcp header and calculate its check sum.

```
tcph->dest = htons ( port );
tcph->check = csum(&psh, sizeof(struct pseudo_header))
```

Listing 1.7: C code to set port and calculate checksum in tcp header

The function sendto() is used to send the created packet to the target machine and port. If the sending fails the program terminates with an error, because then we cannot scan for open ports.

```
if(sendto(s, datagram, packetsize, 0 , &dest, destsize)< 0)
{
perror("Error␣sending␣packet:␣");
exit(0);
}
```

Listing 1.8: C code to set port and calculate checksum in tcp header

To complete the syn port scan, we also have to receive the answer to our sent packet. To receive packets with from a raw socket the function recvfrom() is used. The function call blocks, until it receives a packet from the given socket. Therefore we used the function select, to add an timer to the receiving socket. As you can see in the code of the following listing, we add an timer of 1 sec to the receiving socket and only use the recvfrom() function if the socket contains a packet. If we cannot receive an answer then we simply scan the next port in our implementation.

```
FD_ZERO(&fds);
FD_SET(s, &fds);
tv.tv_sec = 1;
tv.tv_usec = 0;
select(s+ 1, &fds, NULL, NULL, &tv);
if (FD_ISSET(s, &fds))
{
data_size = recvfrom(s, buffer, 65536, 0, &saddr, &saddr_size);
 ..............................
}else{
 printf("Timeout,␣port␣%d␣filtered␣by␣firewall", port);
 return 0;
}
```

Listing 1.9: C code to receive the packet

Because it could be the case that we receive packets from other requests, we first have to check, if the received packet is an answer to our request. To do so we use the IF-Statement of the Listing 1.10. It checks if the source port of the received packet equals the destination port of our sent packet and if the source ip equals to the destination ip to which we sent the packet.

```
if(source.sin_addr.s_addr == dest_ip.s_addr &&
port == ntohs(tcph->source))
```

Listing 1.10: IF statement to check origin of packet

If the received packet passes the check, we know that we have the packet we were looking for. To test if the port is open we finally only have to check if the ack and syn flags are set in the tcp header of the answer. To do so, we first extract the tcp header from our answer by using the length of the ip header as an offset. This works because the first bytes of our answer contain the ip header, which is followed by the tcp header. As it can be seen in the listing 1.11, we finally use an IF statement to check if the tcp header contains the flags which we desire. If it is the case, we know that the scanned port is open.

```
struct tcphdr *tcph=(struct tcphdr*)(buffer + iphdrlen);
```

```
if(tcph->syn == 1 && tcph->ack == 1){
printf("Port: %d is open\n", port);
}
```

Listing 1.11: C code to check if answer contains syn and ack flag

1.3 XMAS, TCP NULL and FIN scan

1.3.1 Theory

1.3.2 Details of implementation

The implementation of the XMAS, TCP NULL and FIN scan is quiet similar to the implementation of the syn scanner. In fact these four scan methods have very much in common. They all need a raw socket to work. There are only 2 main differences between this scan methods and the syn scan. The first one is that XMAS, NULL and FIN scan set different flags in the tcp header. As the name suggests the NULL scan sets none of the flags and the FIN scan only sets the FIN flag. The XMAS scan sets the FIN, PSH, and URG flags, lighting the packet up like a Christmas tree. The second difference of this three scan methods to the syn method is how the answer to the request is used to determine if a port is open or not. If we get a packet with the RST flag as an answer of one of this three scan methods, we know the port is closed. If we don't get an answer we know that the server must have dropped the packet because of an illegal request (RFC 793).

2 Port Scan Detection

Port Scan Detection is fundamental in order to identify potential attackers or to recognize intrusion attempts.

There are many approaches to detect port scans, but the main idea is always the same: *O*ne target is potentially port scanned if it receives from the same source a lot of packets in a short time range. Simple examples of port scan detectors are given by Openwall [**?**] and Sophos [**?**]. We chose the second method because it is described in more details and the document found is more recent.

2.1 Demo Description

2.2 Port Scan Detector

2.2.1 Theory

According to the description of Sophos their port scan detector works as follows [**?**]:

> A port scan is detected when a detection score of 21 points in a time range of 300ms for one individual source IP address is exceeded.

When a packet from one source is received the detection score of this source is actualised adding the points accordingly to the following rules:

- Destination port $<$ 1024: 3 points
- Destination port $>=$ 1024: 1 point
- Destination port 11, 12, 13, 2000: 10 points

The assignments of the points could seem random, but it can be explained by observing the services running on the different ports and the threats associated to the ports.

Score explanation

The Internet Engineering Task Force (IETF) distinguish three ranges of ports: System Ports (0-1023), User Ports (1024-49151) and Dynamic and/or Private Ports (49152-65535) [**?**]. The first range is of particular interest for attackers because it contains many well-known services such as FTP, SSH and HTTP. The other ranges are generally of less interest.

The Ports 11, 12, 13 and 2000 are of particular interest because they are associated with different attacks or can be used to gather information about the victim. According to IANA [**?**] the port 11 is assigned to the service Active Users. If this service is running and receives an UDP or TCP packet it replies with the list of the active users (i.e. the logged users) independently from the content of the packet [**?**]. Port 13 is assigned to the daytime service. If this service is running and receives a TCP or UDP packet it responds with the actual time of day without considering the content of the received packet. Different machines responds with different date/time format, so this can be used to fingerprint the machines [**?**]. Port 2000 is officially assigned to the CISCO SCCP service, but it is also famous for many Trojans such as Der Spaehr, Remote Explorer 2000 among others [**?**]. Port 12 is not associated to any service and attacks but its importance is probably due to the fact that it resides between port 11 and 13.

2.2.2 Details of implementation

The implementation of the Port Scan Detector is largely based on the use of the tcpdump's pcap library and on a good example given by tcpdump self [**?**]. This library is used by tcpdump and wireshark in order to get the packets on the network and choose which packets to sniff and which to ignore by using filter expression such as `dst host 192.168.0.1` and `dst portrange 1-1024` or a combination of those rules [**?**].

In order to start a capture session the program has to create a new pcap object using the `pcap_open_live` function:

```
handle = pcap_open_live(dev, SNAP_LEN, 0, 1000, errbuf);
```

Listing 2.1: Create a new pcap handler.

Another important step is to compile the filter expression.

```
pcap_compile(handle, &fp, filter_expression, 0, net);
pcap_setfilter(handle, &fp);
```

Listing 2.2: Pcap functions called to compile the filter expression and to set it.

A fundamental function is the `pcap_loop`. The parameters are the pcap object, the num of packets (-1 for infinity) a callback function that is called each time a packet is got and the last is usually set as NULL, but it can be used to pass extra arguments for the callback function.

```
pcap_loop(handle, num_packets, got_packet, NULL);
```

Listing 2.3: Pcap functions called to start getting the packets on the network (Error handling omitted).