Portscanner

Mirko Bez Simon Targa

January 22, 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

According to the Techopedia [?] port scanning is defined as follows:

> Port scanning refers to the surveillance of computer ports, most often by hackers for malicious purposes. Hackers conduct port-scanning techniques in order to locate holes within specific computer ports. For an intruder, these weaknesses represent opportunities to gain access for an attack.

1.1 TCP connect scan

This section gives a short overview of the theory behind TCP connect scans and afterwards describes some interesting details about our implementation of this technique.

1.1.1 Theory

The TCP connect scan is probably the simplest 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=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

This section is concerned with the TCP SYN scan. First it describes the theory behind such scans and afterwards follows a description about interesting details of our implementation.

1.2.1 Theory

The TCP SYN scan is often also called half-open-scanning, because it does not open a full TCP connection. The first step of this technique is to send a SYN packet, just like if you are going to open a real connection and then wait for a response. Afterwards the response will be processed. A SYN/ACK means that the port is listening (open), while a RST (reset) will indicate that the port is close. If no response is received the port will be marked as filtered [**?**].

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=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 header, 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, FIN and Maimon scan

This section is about the scan methods XMAS, TCP NULL, FIN and Maimon scan. The first part describes the theory behind them and in the second part we will describe some details of our implementation.

1.3.1 Theory

The XMAS, NULL and FIN scan exploit a subtle loophole found in the TCP RFC 793 to distinguish between open and closed ports. Page 65 of the RFC states that "if the [destination] port state is CLOSED .... an incoming segment not containing a RST causes a RST to be sent in response." The next pages discusses the behaviour if packets are sent to open ports without the SYN, RST, or ACK bits set. It states that: "you are unlikely to get here, but if you do, drop the segment, and return."

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).

1.4 Demo Description

The help page seen in listing gives all necessary information to use the port scanner. It lists all available options and the default values in case the options are not specified.

```
Usage: ./portscanner [options]
Options:

-h   print this help page and exit
-p   specify a port range in form min-max (Defualt 1-1023)
-u   specify an URL to scan (Default localhost)
-v   verbose output
-s   perform a SYN scan
-n   perform a NULL scan
-f   perform a FIN scan
-x   perform a XMAS scan
-m   perform a Maimon scan
```

Listing 1.12: Help page of port scanner

The standard scan method of the port scanner is the TCP connect scan, which will be used, if the user does not select any other method. The option `-p` allows to set the port range to scan. By using the option `-u` the user can

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: One 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 Port Scan Detector

2.1.1 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 received [**?**]. 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.

2.1.2 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 [?]. This library is used by tcpdump and wireshark in order to get the packets on the network. Pcap gives the possibility to 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 and to associate it to the pcap handle.

```
sprintf(filter_expression, "dst␣host␣%s", dst_ip);
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 `pcap_loop`. The parameters are the pcap object, the number of packets (0 or a negative number for infinity), a callback function that is called each time a packet is got and the last argument is usually set as NULL, but it can be used to pass extra arguments for the callback function. Thanks to this function the program enters in a loop and begins to catch packets and process them using 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.

The function `got_packets` is the core of the implementation: Inside this function the packets are processed.

```
void got_packet(u_char *args, const struct pcap_pkthdr *header,
const u_char *packet) {
  ...
  ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
```

The first step is to cast the packet in the structure `sniff_ip` defined in the file `includes/important_header.h`. The source address is read from the packet and used to check if the packet comes from a known source. If it is not the case the source ip is added to the list and a thread is started.

```
  actual_node = contains_node(head, actual_adress);
  if(actual_node == NULL){ //First packet of this source ip
    push(&head, actual_adress);
    actual_node = contains_node(head, actual_adress);
    pthread_create(&actual_node->thread, NULL, thread_function,
            actual_node);
  }
```

According to the packet's protocol the counters of the `actual_node` are actualised and in case of a TCP or UDP packet the function `process_tcp` and `process_udp` are called. Because the UDP callback function is very similar to the TCP one only the latter one is described.

```
  switch(ip->ip_p) {
    case IPPROTO_TCP:
      actual_node->tcp[INDEX_TCP]++;
```

```
      process_tcp(packet, ip, size_ip, actual_node);
      break;
    case IPPROTO_UDP:
      actual_node->udp++;
      process_udp(packet, ip, size_ip, actual_node);
      break;
    case IPPROTO_ICMP:
      actual_node->icmp++;
      break;
    case IPPROTO_IP:
      actual_node->ip++;
      break;
    default:
      actual_node->unknown++;
      break;
  }
```

Listing 2.4: The got_packet function.

The packet is cast to TCP, the destination port is read and the score for the port is saved in the variable tmp.

```
void process_tcp(const u_char *packet, const struct sniff_ip *ip,
  int size_ip, node_t * actual_node){
  ...
  tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
  ...
  dst_port = ntohs(tcp->th_dport);
  int tmp = get_score(dst_port);
```

The score is actualized using the principle described in section 2.1.1

```
int get_score(int dst_port){
  if((dst_port >= 11 && dst_port <= 13) || dst_port == 2000)
    return 10;
  else if(dst_port < 1024)
    return 3;
  else if(dst_port >= 1024)
    return 1;
  return 0;
}
```

Depending on the flags the type of TCP packet is recognized and both the counters for the different packets and the score for the different scan types are actualized.

```
  if(tcp->th_flags == get_syn_scan_flags()){
    index = INDEX_SYN; printf("SYN\n");
  } else if(tcp->th_flags == get_ack_scan_flags()){
    index = INDEX_ACK; printf("ACK\n");
  } else if(tcp->th_flags == get_null_scan_flags()){
    index = INDEX_NULL; printf("NULL\n");
  } else if(tcp->th_flags == get_xmas_scan_flags()){
    index = INDEX_XMAS; printf("XMAS\n");
  } else if(tcp->th_flags == get_fin_scan_flags()){
    index = INDEX_FIN; printf("FIN\n");
  } else if(tcp->th_flags == get_maimon_scan_flags()){
    index = INDEX_MAIMON; printf("MAIMON (FIN | ACK)\n");
  }
  else {
```

```
    index = INDEX_UNKNOWN;
    printf("Not known scan type (Flag set to 0x%X :=", tcp->th_flags);
    print_tcp_flags(tcp->th_flags);
    printf(")\n");
  }
  actual_node->tcp_actual_score[INDEX_TCP] += tmp;
  actual_node->tcp_total_score[INDEX_TCP] += tmp;
  actual_node->tcp_actual_score[index] += tmp;
  actual_node->tcp_total_score[index] += tmp;
}
```

As already mentioned for each different source IP a thread is started. This thread waits 300 milliseconds and then check the actual score for the different TCP/UDP scan types. If the actual score is bigger than 21 a port scan is detected and a briefly message is output.

```
void * thread_function(void * _node){
  node_t * n = (node_t *) _node;
  while(!break_thread){
    nanosleep(&_my_time, NULL); //Wait 300 ms
    controlScore(n);
  }
  return NULL;
}

void controlScore(node_t * n){
  int i;
  int tot_detected = 0;
  for(i = 0; i < INDEX_SIZE; i++){
    if(n->tcp_actual_score[i] >= PORT_SCAN_SCORE){
      printf("*** %7s TCP SCAN FROM %s dectected\n",
            index_to_string(i), n->ip_src);
      n->tcp_scan_detected[i]++;
      tot_detected++;
    }
    n->tcp_actual_score[i] = 0;
  }
  if(n->udp_actual_score >= PORT_SCAN_SCORE){
    printf("=== UDP SCAN FROM %s ===\n", n->ip_src);
    n->udp_scan_detected++;
  }
  n->udp_actual_score = 0;
}
```

In order to stop the loop the function `pcap_breakloop()` has to be called. In the implementation this function is called in the SIGINT (Ctrl-C) handler `my_sigint_handler()` as shown in listing 2.5. This function stop the loop and cause the stop of all the threads.

```
void my_sigint_handler(int d){
  pcap_breakloop(handle);
  break_thread = true;
}
```

Listing 2.5: Handler that process the SIGINT signal.

2.2 Demo Description

In listing 2.6 the options of the port scan detector are described.

```
Usage: ./port_scan_detector [options]
Options:

--help
-h  print this help and exit
--save-the-port
-s  save all the tcp ports requested from the potential attackers
--device
-d  specify a device to use (e.g. wlan0).
    you can get those devices using the command ifconfig
--verbose
-v  verbose output
--dst-ip
-i  give a destination IP adress
--max-num-of-packets
-m  maximum number packets to get, after which the program ends
--log-file
-l  Save the output in a .tex file
```

Listing 2.6: Help page of the port scan detector.

If the user does not specify any option the default behaviour is used: The destination IP is set to 127.0.0.1 and the network interface to lo. Only the packets sent from localhost can be caught and processed.

If the user set the option `-s` the program takes track of how many times each potential attacker which UDP or TCP port visit.

Thanks to the option `-m` the user can set the maximum number of packets to get, after which the programs end.

The option `-l` save all the information in a LaTeX file.

A known issue with the program is that if the user set the option `-i` he or she have also to set the option `-d`.

After the start of the program the information about the single packets and the scan detected are printed on the terminal. The program can be ended either by clicking Ctrl-C or by waiting the reach of the num packets (It can happen only if the option m is set or an error occurred). If the program is ended gracefully it prints a summary of what happened during the capture session.

3 Future Work

There is room of improvement in our project. Section 3.1 gives some advice how the port scanner may be improved and section 3.2 suggests some possible improvements for the port scan detector.

3.1 Port scanner

- Currently in our port scanner we have set a timeout to wait one second for a response. This value could be increased/decreased by doing some tests on different machines and servers to determine a stable value for the timeout.
- Our port scanner loops over every port to scan and only tries once to send a request to the server. If a server is unstable this behavior could led to inconsistent results. It may be better to choose a number of repetition to send the request after getting the timeout.
- Multithreading could be used to scan different ports simultaneously.
- Other scan methods could be added the port scanner (e.g. Idle Scan, UDP scan).

3.2 Port scan Detector

- The port scan detector currently uses only one network interface. It may be also interesting to have an option to use more network interfaces simultaneously.
- Currently the detector takes the IP address of the interface as a command line argument. In order to make it more user-friendly the IP address could be determined at runtime.
- In order to distinguish between a TCP connect and SYN scan techniques could be implemented to recognize a complete handshake.
- Implementation of other port scan detection approaches.