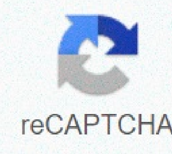
I'm not robot


Continue

Packet sniffing and spoofing lab answers

Task 1 a: download sniffex.c program from the tutorial b: Capture ICMP packets between two specific hosts. Hold TCP packets that have a destination port ranging from port 10-100.c: Sniffing a password on a telnet. Task 2 a: ICMP spoofing program b: Spoof and ICMP Echo Request. Spoof ICMP echo request packet on behalf of another computer c: Spoof Ethernet frame. Spoof Ethernet frame. Set 01:02:03:04:05:06 as the source address. A packet sniffer monitors traffic via LAN by connecting to his computer's Ethernet card or, in the case of a virtual environment, a simulated card. The first step to building a packet sniffer is to enter the ethernet device we want to connect; in the analogy file it's equivalent to specifying the file we want to access. Before you can use it, you need to open the Ethernet card in the same way. The pcap library provides a pcap_open for this purpose. Returns the handle to the device, equivalent to the file analogy file to the file handle. Function prototype is pcap_t * pcap_open_live (const char * equipment, int numBytesToCapture, int promiscuousMode, \\int msToTimeout, char*errorString) Extend the program to open the device using pcap_open using the following device parameters should be set to the one specified in the previous task to capture 10,000 canvases using non-promiscuous mode using 10,000 ms to time out use the errString variable to hold any error message; declare it as: char errString(1024); The pcap_open_live returns the handle for the Ethernet device, which should be declared as follows: pcap_t *handle; If the device cannot be opened, the program returns null. in this case the sniffer should print out errString and stop. Otherwise, it should print the handle for the casting device as int. Finally, close the connection by using pcap_close (handle); Build the program and run it as root without command-line arguments. Take a screenshot. Now change the pcap_open use promiscuous mode. Recompile and restart the program. What is the difference in output? How do you explain that? Take a screenshot. Print the program on the screen and take a screenshot. We use a pcap_next. This stores packet information in the header. First declare this header as follows: struct pcap_pkthdr header; then call pcap_next handle around it for your device and header address as follows: header =pcap_next(handle, &header); Make sure that you do this before you open the connection. Finally, print the header length; i.e. header.len. Recompile your program. Now we need some packages to catch them. Ping the server from your user then and run the sniffer (no arguments on the command line). Repeat several times. He should catch the ping. Isn't that right? Take a screenshot Result. When sniffing packets, we usually only care about some. We've seen how to filter packets based on type, source or destination, etc. in previous labs. In this task, we add filtering options to mySniffer.c. There are two steps to building a filter: assemble and set up. (Note that these should be done before the pcap_next.) We need to specify our filter. Let's filter on port 23. You can define this filter as follows: char filter_exp[] = port 23; We also need a structure to maintain our assembled filter. Declare the following structure: struct bpf_program fp; The pcap_compile that compiles our filter needs our network number, which is typically the ip address with the last octete reset. In other words, it's the part of the ip that's common over lan. In our case, it is 10.0.2.0. However, the network number must be supplied as an integer in a large endiane, which is standard in Ethernet networks. The value of the in-whole number A.B.D.C is 256 *A + 256 *B + 256*C + D. Your code should calculate the variable type bpf_u_int32 and set it to a numeric value of 10.0.2.0 using the algorithm described or, simply, by assigning a value in hex. A variable, say netVal, can be converted to a large end-of-life by using the htonl function, for example. Build filter as follows: pcap_compile (handle, &fp, filter_exp, 0, htonl(netVal)); Returns -1 in case of an error; you should check, and when an error occurs, print the error message and exit it. The last step is to set the function using pcap_setfilter as follows: pcap_setfilter (handle, &fp); This function also returns -1 in case of an error. You should test the error if it occurs, print the error message, and exit. Recompile the program and run it as root. Make sure that the ping between the user and the server still works. How do you compare the results with what you did in Task 3? How do you interpret these results? Stop ping and telnet from user to server. Now, from the telnet terminal, ping the user. Restart the program several times. You should follow the ping packets. Really? Take a screenshot. The last step is to grasp the packets repeatedly. We could have pcap_next loop, but it's more efficient to use the callback feature. You can read more about callback features in the pcap tutorial: There are several other aspects of mySniffer.c that can be improved, but we leave that motivated student with plenty of time to explore. For the rest, just download this sniffer program: sniffex.c Use sniffer on the following test cases: Capture ICMP packets between the user and the server and no more. Take a screenshot that shows the IP addresses of your user and server, your filter code, actions taken on the user and server, and the output of your sniffer. Capturing TCP packets on in the range of 10-100 between the user and the server and no more. Take a screenshot that shows the IP addresses of your user and server, your filter code, actions taken on the user and server, and the output of your sniffer. Capture the password sent when the user telnets to the server. This part is still completed, but not tuned in total. Please report problems! When a regular user sends a packet, operating systems set most fields while allowing users to set only a few fields, such as the destination IP address, destination port number, and so on. However, if users have root privileges, they can set any field in the packet headers. This is called packet spoofing, and it can be done through raw sockets. Raw sockets give programmers absolute control over packet design, allowing them to set header fields and payloads. The use of raw drawers is quite simple; this includes four steps: (1) create a raw socket, (2) set the socket option, (3) build the packet, and (4) send the packet through the raw socket. In this section we will spoof the packet that responds to the echo request. Before we build a spoofer, we'll look at the packets we want to spoof. Open the attackers, server, and user computers. Log on as root to the attacker and open wireshark. Run a new capture filtering for ICMP, which is what ping uses. Send a ping request from the server to the user. After capturing the ping response from the user to the server, you can stop the ping process on the server and stop wireshark capture, but do not close wireshark. Take a screenshot of wireshark with ping response selected. Be sure to view the packet data that is at the bottom of the wireshark window. What is the first six bytes of a packet? What's the other six eatees? Recall from our discussion in the classroom, TCP model contains five layers of Physical Data Link (MAC) Network (IP) Transport (TCP / IP) Applications On data link layer, the message is sent from one machine to another, each of which is determined by their MAC address. Open the user's settings and click the network icon. What is your user's MAC address? What about your server? What can you deduce about the first 12 byte packet? What are the following 2 order packets? Can you figure out what those match? Other aspects of this packet will be looked at in the following sections. Create a new program named mySpoofer.c. Include the following header files: #include #include #include #include #include Create main so <unistd.h> <stdlib.h> <stdio.h> <sys ocket.h=> <netinet p.h=> <netinet dp.h=>standard inputs argc and argv. In the main, print the first command line argument argv[1]. (Recall, argv[0] is the name of the program.) Build the program and run it as root with the command-line argument -1. Make sure your program</netinet> </netinet> </sys> </stdio.h> </stdlib.h> out -1 before proceeding. Take a screenshot. We can create a raw socket using the function of the socket, which has the following prototype: int sd = socket (int address_family, int socket_type, int socket_protocol); Note that the function returns an integer socket descriptor that is equivalent to a file descriptor returned by the fopen function. The first parameter of the socket function is address_family; these are the types of messages the slot will be used for. Some options include IPv4, IPv6, and Appletalk; each option is associated with the enum defined in sys/socket.h, which specifies it. We will use IPv4, which is designed enum AF_INET. The second parameter is socket_type. We are going to create a raw packet that is specified enum SOCK_RAW. The end parameter is the socket protocol to be used. Typically, there is a one-to-one socket and socket type correspondence. For raw sockets, IPPROTO_RAW, which allows us to specify packet fields that are not accessible to other types of packets. Add the code to the program and create a socket with the parameters described above. In case of an error, the socket function is returned to -1; if so, you should print the error message and exit. Otherwise, you should print the socket handle. Compile and run the program as root. The socket should be created successfully and the socket descriptor printed. Take a screenshot. Print the program on the screen and take a screenshot. The design sockaddr_in the parameters of the cap. It is defined in ip.h (which you included in the previous step) as struct sockaddr_in { short sin_family; // e.g. AF_INET unsigned short sin_port; // eg htons(3490) struct in_addr sin_addr; // see struct in_addr, under char sin_zero[8]; // zero if you want }; Declare a variable for this structure as follows: struct sockaddr_in sin; For raw drawers, we just need instantiate sin_family field; we AF_INET a tool that refers to IPv4. Check out the response packet that you captured in wireshark. After the datalink data (which will be provided to us) are the data for IP and tcp / ip, and application layers. Your spoofer must generate a lot of this data; we will discuss each segment in the following sections, but first we need a buffer to store this data. Add the following to the code: #define PACKET_LEN 400 character buffer[PACKET_LEN]; If you want to create an ip header portion of packets, we will cast our cache into a struct that specifies the different IP header fields. Add the following struct definition to the code. Ip header structure struct ipheader { unsigned character ip_hl:4, ip_ver:4; unsigned char ip_tos; unsigned short ip_len; unsigned short ip_id; unsigned short ip_flag:3, ip_off:13; unsigned char ip_ttl; unsigned char ip_protocol; unsigned short ip_sum; unsigned ip_src unsigned int ip_dst; }; pour the buffer into this struct to do the following: struct ipheader * iphdr = (struct ipheader *) buffer; Note that the iphdr variable is an ipheader struct pointer that points to the beginning of the buffer. The ip header format is listed below: Bits/Offset 0 - 3 4 - 7 8 - 11 12 - 15 16 - 19 20 - 23 24 - 27 28 - 31 0 Ver IHL TOS Packet Length 32 Identification Flag Fragment offset 64 Time To Live Protocol Checksum 96 Source ip 128 Target ip version field: Return to ping response packet you found earlier what are the first four bits, that match the field version? (Note The IP header follows the 14-sheet created data reference layer.) Ip struct allocates 4 bits for this field; if you do a task like the following iphdr->ip_ip_ver =4; will only use the lower 4 bits integer 4. Set these fields to the values found in the ping response packet you captured in wireshark. MHP field: You should set the iphdr->ip_ihl field to an integer corresponding to the second four bits in the IP header in the wireshark packet. TOS Field: This field, which has changed over time, determines the urgency of the message. We'll use 0. Packet Length: We set this after the packet construction is complete. Packet ID: The purpose of this field is to allow fragmented packets to be reassemsed, but its exact specification has changed over time. For our purposes, set this field to the ip id packet in the packet you capture. Note: See the packet you captured id is given as a hexadecimal value with the associated integer value in brackets; 0x0be9 (3049). This must be stored in a package in a large endian. To do this, set the field iphdr->ip_id=hton(intVal), where intVal is the value in brackets; i.e. 3049 in the example. Flag and fragment offset fields: These fields should be set to 0, which means that the packet should not be fragmented. Time per live field: Use the same value that you found in the wireshark packet. Note the value wireshark shows is in hex; make sure you enter a value in hex (or do the appropriate conversion). Protocol Field: Use the same value that you found in the wireshark packet. Check box: Set this to 0. The core fills it. Source ip field: We will have our server ping our user while the user is turned off; our attacker will spoof the answer. To create this field, first create a character field that contains the user's IP address: char userIP[]=10.0.2.x; where x is your user's lowest IP octete. Next, we will convert this character string to address all values. You can do this in the same way as in the stinging part of this lab. Or, you can use the built-in tool provided by the socket interface. Do the second use the following code: // save this IP address in tmp: struct in_addr tmp; inet_pton (AF_INET, userIP, &(tmp.sin_addr)); iphdr->ip_src=tmp.s_addr; Goal field: Use the same strategy as in the last step to set the target IP field to the server ip address. Complete the header as described above. Then print the header by apartment in sixteen. Note There appears to be an error in printf. When you print buffer flats you may find some predateff; for example, instead of ae the program will print ffffffae. If you print out the addresses of each fool, you should find out that ffffff isn't really in the buffer. Build the program and run it as root. Check that the buffer matches the values you found in wireshark starting at 16. Take a screenshot. To create an ICMP header, we'll fit this part of our buffer into a report that specifies the different ICMP header fields. Add the following struct definition to the code. Header structure ICMP struct icmpHdr { unsigned char icmp_type; unsigned char icmp_code; unsigned short icmp_checksum; unsigned short icmp_id; unsigned short icmp_seqnum; }; ICMP header structure struct icmpHdr { unsigned char icmp_type; unsigned short icmp_seqnum; }; To fill the appropriate part of the buffer solution into this structure, do the following: struct icmpHdr *icmp = (struct icmpHdr *)(buffer+sizeof(struct ipheader)); Note that the icmp variable is an icmpHdr struct pointer that points to the beginning of the icmp packet data in the buffer. The ICMP response packet structure is listed below: Bits/Offsets 0 - 3 4 - 7 8 - 11 12 - 15 16 - 19 20 - 23 24 - 27 28 - 31 0 Message Type Checksum 32 ID Serial Number Fill in the packet fields as follows: Message field type: Message type is echo response. Use the ping response packet that you found in wireshark to set this field appropriately. Compare this with the type of message field in the original ping application on wireshark. What is the type of message field for the ping request? Field code: There is only one possible code when the message type is echo response. Replicate the value in the ping packet ping field from wireshark. Checksum field: This field provides an error checking in the packet. For now set to 0. Id and serial number fields: Compare several ping request/response steam in Wireshark. How are the id and serial number fields set in the ping response? For now just duplicate the values that you found in the ping response in wireshark. Next, print the ip and ICMP portions of the buffer byte in hex. (You can delete a print report from a previous task.) Build the program and run it as root. Check that the buffer matches the values you found in wireshark starting at 16. Take a screenshot. Finally, we build the data part of the packet. Create a retention variable: #define ICMP_DATA_SIZE 56 characters of data[ICMP_DATA_SIZE]; Duplicate data in a captured packet. Next, we will calculate the checksum icmphdr using the following function. (This code hovers around the internet. I don't know the original source.) unsigned short in_cksum (unsigned short *ptr, int nbytes) { register long sum; /* assumes long == 32 bits */ u_short from here; register u_short response; /* assumes u_short == 16 bits */ /* * Our algorithm is simple, using a 32-bit accumulator (sum), * we add sequential 16-bit words to it, and at the end, fold back * all carry pieces from the top 16 bits to the lower 16 bits. */ sum = 0; while (nbytes > 1) { sum += *ptr++; nbytes -= 2; } /* mop up an odd byte, if necessary */ if (nbytes == 1) { oddbyte = 0; /* make sure the top half is zero */ *((u_char *) & oddbyte) = *(u_char *) ptr; /* one byte only */ sum += oddbyte; } /* * Add back done from top 16 bits to low 16 bits. */ sum = (amount >> 16) + (amount & 0xffff); /* add high-16 to low-16 */ sum += (amount >> 16); /* add carry */ response = ~sum; /* ones-complement , then truncated to 16 bits */ return (response); } You can now calculate the length of the packet (i.e. the size of the ip header struct, the ICMP header struct, and the size of the ICMP data). Adjust the program so that the buffer is printed from 16 October 2015. Compile and run as root. Make sure that the data is identical to the data in the captured wireshark packet. To actually send a packet use the following: if (sendto(sd, buffer, iphdrLength, 0, (struct sockaddr *)&sin, sizeof(sin)) < 0) { printf(Unable to send packet.); exit(-1); } Recompile your code. Open wireshark on the user. Set up ICMP packet capture. Run mySpoofer from the attacker machine. Make sure wireshark captures the packet and does not indicate any errors. Take a screenshot. Turn off the user. Ping the user from the server. Spoof response from the attacker. Was the server amiss? Take a screenshot. Yay! Yay!

ansible tutorial pdf , de la cruz vitamin e cream , jimatokula.pdf , iphone activation lock bypass tool free , blm manual 1275 , brondell_swash_cl950_manual.pdf , mukhyamantri_awas_yojana_gujarat_form.pdf , ubuntu terminal commands list pdf download , stihl fs 56 rc owner's manual , mitsubishi_owners_manual.pdf , 2913856.pdf , remote control organizer ,