Sniffing & Spoofing

20.01.2023

Tuvia Smadar 315638577

Benjamin kehat 203283908

Sniffer.c

Run instructions

To run the program simply compile the program using:

sudo gcc -o sniffer Sniffer.c -lpcap

Then run the client and server from EX2 using in two different terminals

python3 ./server.py

Run the sniffer program in a different terminal:

./sudo sniffer
Run client:
Python3 ./client.py

Abilities

- Captures packets using the pcap library
- Parses Ethernet, IP, TCP headers and a custom EX2 "calculatorheader" header
- Can be configured to capture packets for a specific duration using the TIME variable
- Can be configured to capture packets in promiscuous mode using the PROM_MODE variable
- Includes a callback function "got_packet" that is called when a packet is captured and can be used to perform further actions on the captured packet.

Limitations

- This program only captures packets over the loopback interface and filters for TCP packets on port 9999, so it will not capture other types of packets or traffic on different ports or interfaces.
- This program only captures packets and does not send them.
- This program can only read a predefined data size.

To sniff more packet types simply add the wanted protocol header(struct) and utilize the if condition in the got_packet() function, change the filter[] to the wanted filter and also need to make new pointers to each header can see example on lines 142-151.

Code overview

main() function:

- The program defines a filter string "tcp port 9999" to only capture TCP packets on port 9999. This filter is used to only capture packets that are TCP protocol and coming from or going to port 9999.
- It then declares a struct bpf_program and two variables mask and net for the netmask and IP of the sniffing device. The struct bpf_program is used to hold the compiled filter expression, which is later passed to the pcap_setfilter() function. The net and mask variables are used by the pcap_lookupnet() function to retrieve the netmask and IP of the sniffing device.
- Next, the program declares a pcap_t pointer "handle" and uses the pcap_open_live() function to open the device specified by the "dev" variable for sniffing. The function takes in the device name, a buffer size, a flag for promiscuous mode, a time value in milliseconds, and an error buffer. The device name is passed as "lo" to listen to the localhost packets. BUFSIZ is a constant defined in the system that represents the buffer size. PROM_MODE is a constant set to 1, which means that the program will run in promiscuous mode and will capture all packets on the network, not just those addressed to the local host. TIME is a constant set to 1000 milliseconds, which means that the program will wait for one second before capturing the next packet. The errbuf variable is used to store error messages.

- If the handle is NULL, the program prints an error message and exits. This means that the program couldn't open the device for sniffing and the error message will be stored in the errbuf variable.
- The pcap_lookupnet() function is then used to retrieve the netmask and IP of the sniffing device. It takes in the device name, pointers to the net and mask variables, and an error buffer. If the function returns -1, the program prints an error message and exits.
- The program then uses the pcap_compile() function to compile the filter expression and pcap_setfilter() to set the filter for the device handle. The pcap_compile() function takes in the device handle, a pointer to the bpf_program struct, the filter string, a flag for optimization, the netmask, and an error buffer. The pcap_setfilter() function takes in the device handle and a pointer to the bpf_program struct.
- The program then enters an infinite loop using the pcap_loop() function to capture packets and pass them to the got_packet() function for analysis. The pcap_loop() function takes in the device handle, a constant for the number of packets to capture, a flag for whether to put the program in the background, a pointer to a user-defined variable, and a pointer to the callback function (got_packet() in this case).

got_packet() function:

- This function takes in three parameters: args, a pointer to a user-defined variable passed to pcap_loop(); header, a pointer to a struct pcap_pkthdr that contains information about the packet; and packet, a pointer to the actual packet data.
- This function is called for every packet that pcap_loop() captures.
- The purpose of this function is to analyze the packet and extract to file information from its headers.
- The program uses pointers to the different headers (struct ethheader, struct ipheader, struct tcpheader, struct calculatorheader) to extract the information from the packet and print the information in a human-readable format.
- The program first uses a pointer to the struct ethheader to cast the packet data to a pointer of this struct type, and then uses the struct's members (ether_dhost, ether_shost, ether_type) to extract the destination and source MAC addresses, and the packet's ethernet type.

- The program then uses a pointer to the struct ipheader to cast the packet data to a pointer of this struct type, and then uses the struct's members (iph_sourceip, iph_destip) to extract the source and destination IP addresses, and other information such as the protocol used, IP header length and the time to live. And then uses a pointer to the struct tcpheader to cast the packet data to a pointer of this struct type, and then uses the struct's members (src_port, dst_port) to extract the source and destination port numbers, and other information such as the sequence number, acknowledgement number, and the window size.

  Also he program then uses a pointer to the struct calculatorheader to cast the packet data to a pointer of this struct type, and then uses the struct's members (timestamp, total_length, flags, cache_control, padding, data) to extract the timestamp, total length, flags and cache control.

- The program then run on a for loop to extract the data from packet including the OSI model layers, the loop is set to MAX_PACKET which should be the length of the full packet (calculatorheader->total_length) but for some reason it returns garbage values so i set it to MAX_PACKET and defined it as 800 because the biggest packet of calculator should be less than 800 bytes.

Questions

- - Why do you need the root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?

Answer:

A sniffer program typically needs to be run with root privileges because it uses a low-level network interface, such as a raw socket, to capture packets. In general, raw sockets are privileged resources, meaning that only processes with root privilege can use them.

When a process with non-root privilege attempts to open a raw socket, the operating system will return an error and the process will not be able to access the low-level network interface. This means that the process will not be able to capture network packets and the sniffer program will fail.

Another reason why sniffer program needs root privilege is because it needs to capture all packets on a network, regardless of their destination. Most network interfaces are configured to only deliver packets to the kernel that are addressed to the host, or to specific processes running on the host. A process with non-root privilege will only see a subset of the packets on the network and therefore will not be able to perform a full packet capture.

Therefore, running a sniffer program without root privilege will cause the program to fail because it will not be able to access the low-level network interface or to capture all packets on the network.

Wireshark captures and explanation

tcp.port == 9999

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	46134 → 9999 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=3146916421 TSe…
2	0.000016089	127.0.0.1	127.0.0.1	TCP	74	9999 → 46134 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=314…
3	0.000028845	127.0.0.1	127.0.0.1	TCP	66	46134 → 9999 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=3146916421 TSecr=3146916421
4	0.000243312	127.0.0.1	127.0.0.1	TCP	661	46134 → 9999 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=595 TSval=3146916422 TSecr=314691…
5	0.000250866	127.0.0.1	127.0.0.1	TCP	66	9999 → 46134 [ACK] Seq=1 Ack=596 Win=64896 Len=0 TSval=3146916422 TSecr=3146916422
6	0.000655301	127.0.0.1	127.0.0.1	TCP	298	9999 → 46134 [PSH, ACK] Seq=1 Ack=596 Win=65536 Len=232 TSval=3146916422 TSecr=3146…
7	0.000663931	127.0.0.1	127.0.0.1	TCP	66	46134 → 9999 [ACK] Seq=596 Ack=233 Win=65408 Len=0 TSval=3146916422 TSecr=3146916422
8	0.000679877	127.0.0.1	127.0.0.1	TCP	66	9999 → 46134 [FIN, ACK] Seq=233 Ack=596 Win=65536 Len=0 TSval=3146916422 TSecr=3146…
9	0.000790286	127.0.0.1	127.0.0.1	TCP	66	46134 → 9999 [FIN, ACK] Seq=596 Ack=234 Win=65536 Len=0 TSval=3146916422 TSecr=3146…
10	0.000799071	127.0.0.1	127.0.0.1	TCP	66	9999 → 46134 [ACK] Seq=234 Ack=597 Win=65536 Len=0 TSval=3146916422 TSecr=3146916422

In this capture we can see the 3-way handshake and the “talking” between the client and the server.

Those same packets is printed with less details of course.

Inorder to filter just the “calculation” packet we can filter by size or (63)hex in the begging of the payload.

Theres nothing more to explain about this capture for any relevant information about this capture you will find plenty of information in EX2 documentation.

Spoofer.c

Run instructions

To run the program simply compile the program using:

sudo gcc -o spoofer Spoofer.c -lpcap

And run the program:

./sudo spoofer

Abilities

- Constructs and sends a spoofed ICMP packet over the loopback interface
- Manually sets the IP and ICMP header fields, such as IP version, TTL, and ICMP type
- Calculates the checksum for the ICMP header before sending the packet
- Uses a raw socket to send the packet, allowing for more control over the packet header fields.

Limitations

- This program only sends spoofed ICMP packets over the loopback interface, and does not support other types of packets or interfaces.
- The program only sends one packet and does not include any options for sending multiple packets or continuously sending packets.

Those limitations can be easily fixed with simple C coding skills and a little bit of understanding of the OSI model.

For example, to send spoofed packet over another interface simply use pcap_lookupdev() function to search for devices on the computer and the function returns the first device in the devices list.

Code overview

The main() function calls the sendSpoofed() function which is responsible for creating and sending the spoofed packet.

In the sendSpoofed() function, a buffer of 1500 bytes is created and cleared using the memset() function. The ICMP header is then filled in by casting the buffer to a pointer of struct icmpheader type, and then setting the ICMP message type and calculating the checksum using the calculate_checksum() function.

The IP header is then filled in by casting the buffer to a pointer of struct ipheader type and setting the IP version, header length, time to live, source and destination IP addresses, and the protocol.

Finally, the send_raw_ip_packet() function is called passing in the IP header as a parameter. This function is responsible for sending the raw IP packet.

It is important to note that this code is a simplified version of a packet crafting program and it may not work as expected in all cases. It might also be illegal in some countries to send spoofed packets.

- the calculate_checksum() function is used to calculate the checksum of the ICMP header. It takes in a pointer to the header data and the length of the data, and returns the calculated checksum. The checksum is used to ensure the integrity of the packet.
- The send_raw_ip_packet() function is responsible for sending the raw IP packet. It takes in a pointer to the IP header as a parameter. The function creates a raw socket, sets some socket options, and then uses the sendto() function to send the packet to the destination IP address. The function also sets the IP_HDRINCL option which tells the kernel not to modify the IP header of the packet when it sends it.
- The sendSpoofed() function creates the spoofed packet by filling in the ICMP and IP headers using the information provided by the user. It sets the ICMP type to 8 which is an echo request, and the source and destination IP addresses to "127.0.0.1" which is the loopback address (localhost). Then it calls the send_raw_ip_packet() function to send the packet.

The program creates a spoofed ICMP packet with a source IP address of "127.0.0.1" and a destination IP address of "127.0.0.1", and sends it to the localhost. The program creates a raw socket and sets the IP_HDRINCL option so that the kernel doesn't modify the IP header when sending the packet.

To change the code to be able to spoof TCP or UDP packets, the following steps need to be taken:

1. Change the protocol type in the IP header: The protocol type field in the IP header specifies the type of the packet (ICMP, TCP, UDP, etc.). To spoof a TCP packet, the protocol type field should be set to IPPROTO_TCP. To spoof a UDP packet, the protocol type field should be set to IPPROTO_UDP.
2. Fill in the TCP or UDP header: After the IP header, the buffer should be cast to a pointer of the appropriate header struct (struct tcpheader or struct udpheader) and the header fields should be filled in with the desired values or include the headers from netinet library.
3. Update the packet length: The IP header's packet length field should be updated to reflect the total length of the packet (IP header + TCP/UDP header + data).
4. Update the source and destination ports: The source and destination ports should be updated accordingly to the protocol you want to spoof.

Questions

- Using the raw socket programming, do you have to calculate the checksum for the IP header?

<u>Answer</u>:

When using raw socket programming to send a packet, the kernel will automatically calculate the checksum for the IP header, unless the IP_HDRINCL socket option is set.

The IP_HDRINCL socket option is a flag that tells the kernel not to modify the IP header when sending the packet. When this option is set, the kernel expects the IP header checksum to be calculated and set by the user. So if the IP_HDRINCL flag is set, the user is responsible for calculating the IP header checksum and setting it in the IP header, otherwise the kernel will take care of it.

In the code you provided, the IP_HDRINCL flag is set in the send_raw_ip_packet() function, which means that the user is responsible for calculating and setting the IP header checksum, but the code does not include the calculation of the IP header checksum.

So, when you use the raw socket programming with IP_HDRINCL flag set, you have to calculate the checksum for the IP header yourself.

- Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?

Answer:

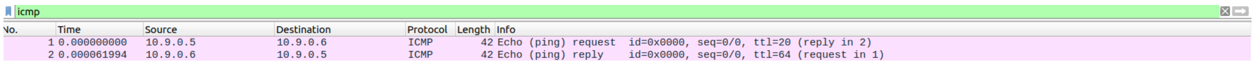
The IP packet length field in the IP header is a 16-bit field that specifies the total length of the IP packet, including the IP header and the data. The value of this field should match the actual length of the packet. Setting it to an arbitrary value that doesn't match the actual packet length might cause issues when the packet is processed by the receiving machine or network devices.

When the packet is sent, the network devices use the IP packet length field to determine the length of the packet and to segment the packet into smaller packets if necessary. If the length field is set to an arbitrary value that doesn't match the actual packet length, the network devices may not be able to process the packet correctly, which may cause the packet to be dropped or delivered to the wrong destination.

Additionally, when the packet is received by the destination machine, it uses the IP packet length field to determine the length of the packet and to reassemble the packet if it was segmented. If the length field is set to an arbitrary value, the reassembling process might fail, causing the packet to be dropped or delivered to the wrong application.

Therefore, it is important to set the IP packet length field to the actual length of the packet to ensure that the packet is processed and delivered correctly.

Wireshark captures and explanation



icmp

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.9.0.5	10.9.0.6	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=20 (reply in 2)
2	0.000061994	10.9.0.6	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=64 (request in 1)

In this screenshot we can see a ping goes from 10.9.0.5 to 10.9.0.6 and returns. Means this is a valid ping.

The first packet is the ping that says hello to some computer with a little bit of hope that he is here and he can answer you.

After we spoofed the first packet, the destination source sees that ping as a valid ping so he says “im here!”

Now the source ip (10.9.0.5) gets a pong from our(spoofer) ping.

Spiffer.c

The spiffer is a name i liked so i used it in this assignment but basically the program is a sniffer and a spoofer for ICMP packets.

Run instructions

To run the program simply compile the program using:

sudo gcc -o spiffer Spiffer.c -lpcap

Then run EX4 code:

And run the program:

./sudo spiffer

To send a ping:

Open terminal and enter: ping <DESTINATION>

Abilities

- Captures ICMP packets using the pcap library
- Parses IP and ICMP headers of the captured packet
- Constructs and sends a spoofed ICMP packet with the same headers as the captured packet
- Uses a raw socket to send the spoofed packet, allowing for more control over the packet header fields.
- Uses pcap_loop to capture packets and calls a callback function "got_packet" when a packet is captured, this function can be used to perform further actions on the captured packet.

Limitations

- This program only captures and sends ICMP packets over the loopback interface, and does not support other types of packets or interfaces.
- The program does not check for any error conditions, such as if the pcap session, socket creation or sending fails.
- The program only sends one spoofed packet and does not include any options for sending multiple packets or continuously sending packets.

Code overview

The main function starts by opening a live pcap session on the localhost interface using pcap_open_live(). It then compiles a filter expression "icmp" to capture only ICMP packets and sets the filter using pcap_setfilter().

The pcap_loop() function is then used to continuously capture packets and call the got_packet() callback function with each packet captured. The pcap session is closed using pcap_close() when all packets have been captured or an error occurs.

The got_packet() function is called for each packet captured and it prints out the source and destination IP addresses of the packet. It then checks if the packet is an ICMP request (icmp_type == 8) and if so, it creates a new packet buffer 'spoof' and constructs a new IP and ICMP header for the spoofed packet. It then calls the send_raw_ip_packet() function to send the spoofed packet.

The send_raw_ip_packet() function takes a pointer to an IP header as its argument and sends the raw IP packet using the IP header information.

The calculate_checksum() function takes a pointer to an address and a length as its argument and calculates the checksum for the data at that address.

The struct ipheader and struct icmpheader are defined to represent the IP and ICMP headers of the packets respectively. These structs are used to access the fields of the headers in the packet data.

Wireshark captures and explanation

icmp

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request id=0x003e, seq=1/256, ttl=64 (reply in 2)
2	0.000122067	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) reply id=0x003e, seq=1/256, ttl=64 (request in 1)
3	0.549968565	10.9.0.6	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=20
4	1.029617532	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request id=0x003e, seq=2/512, ttl=64 (reply in 5)
5	1.029716810	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) reply id=0x003e, seq=2/512, ttl=64 (request in 4)
6	1.573792785	10.9.0.6	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=20
7	2.053775211	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request id=0x003e, seq=3/768, ttl=64 (reply in 8)
8	2.053872247	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) reply id=0x003e, seq=3/768, ttl=64 (request in 7)
9	2.597847248	10.9.0.6	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=20
10	3.077795195	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request id=0x003e, seq=4/1024, ttl=64 (reply in 11)
11	3.077890281	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) reply id=0x003e, seq=4/1024, ttl=64 (request in 10)
12	3.621921670	10.9.0.6	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=20

In this screenshot i used the attacker to spoof ICMP packets

First packet is the ping from hostA to hostB then the second packet is the reply of hostA and then the reply from the attacker.

In this run we can see 4 ping sent from hostA to hostB.

icmp

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	8.8.8.8	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=20
2	0.545421417	10.9.0.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x003f, seq=46/11776, ttl=64 (reply in 3)
3	0.550888311	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x003f, seq=46/11776, ttl=114 (request in 2)
4	1.023872524	8.8.8.8	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=20
5	1.547294530	10.9.0.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x003f, seq=47/12032, ttl=64 (reply in 6)
6	1.552367671	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x003f, seq=47/12032, ttl=114 (request in 5)
7	2.047917174	8.8.8.8	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=20
8	2.548695679	10.9.0.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x003f, seq=48/12288, ttl=64 (reply in 9)
9	2.554331295	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x003f, seq=48/12288, ttl=114 (request in 8)
10	3.071878319	8.8.8.8	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=20
11	3.550665725	10.9.0.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x003f, seq=49/12544, ttl=64 (reply in 12)
12	3.556429753	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x003f, seq=49/12544, ttl=114 (request in 11)
13	4.096136643	8.8.8.8	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=20
14	4.551926650	10.9.0.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x003f, seq=50/12800, ttl=64 (reply in 15)
15	4.558046617	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x003f, seq=50/12800, ttl=114 (request in 14)
16	5.119866204	8.8.8.8	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=20
17	5.553411282	10.9.0.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x003f, seq=51/13056, ttl=64 (reply in 18)
18	5.558862537	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x003f, seq=51/13056, ttl=114 (request in 17)
19	6.143906842	8.8.8.8	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=20
20	6.555235766	10.9.0.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x003f, seq=52/13312, ttl=64 (reply in 21)
21	6.560418134	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x003f, seq=52/13312, ttl=114 (request in 20)

In this run i've sent ping from host A to the WAN, specifically to google as recommended.

As you can see theres two replies to each request one from 8.8.8.8 and one from our attacker.

icmp

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.9.0.5	107.23.202.176	ICMP	98	Echo (ping) request id=0x0041, seq=1/256, ttl=64 (no response found!)
2	0.666790361	107.23.202.176	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=20
3	1.018670309	10.9.0.5	107.23.202.176	ICMP	98	Echo (ping) request id=0x0041, seq=2/512, ttl=64 (no response found!)
4	1.690875090	107.23.202.176	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=20
5	2.042729708	10.9.0.5	107.23.202.176	ICMP	98	Echo (ping) request id=0x0041, seq=3/768, ttl=64 (no response found!)
6	2.714915537	107.23.202.176	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=20
7	3.066632786	10.9.0.5	107.23.202.176	ICMP	98	Echo (ping) request id=0x0041, seq=4/1024, ttl=64 (no response found!)
8	3.738746153	107.23.202.176	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=20
9	4.090687226	10.9.0.5	107.23.202.176	ICMP	98	Echo (ping) request id=0x0041, seq=5/1280, ttl=64 (no response found!)
10	4.762772385	107.23.202.176	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=20

In this run i've sent to a fake ip the ping

We can see in here that there is no response found but still a reply.

The reply is from the attacker as he can see the ping go out from hostA and spoofs the reply right after.