Question 1: TCP Attacks (48 marks)

In this course we began by stating how adversaries can inject, modify, eavesdrop, and delete messages sent over an insecure channel. When we looked at how this can be done for TCP connections we see that there are few more complications (e.g., whether or not an adversary is "on-path"). Now we take this one step further and actually perform these attacks by crafting custom packets and injecting them onto a network.

For these exercises you are required to use the tinycore virtual machine that you have been using in previous exercises. `N.B. Do not use your own machines for this:` there are number of operating system protections that exist that may prevent it from working. There are three submissions for each exercise: (i) a packet capture file ('pcap') that exhibits your attack in action, (ii) answers to questions about that pcap file related to aspects of the attack, and (iii) submission of the code to produce raw packets. All three are to be submitted to the D2L dropbox.

Sending raw packets is tricky and there is often little indications of why it isn't working becuase ill-formed packets are simply rejected by the operating system. You have to set all the TCP and IP header values correctly. You need to set the IP header include socket option (`IP_HDRINCL`). You need to compute checksums. You need to worry about endianness. This includes both the endianness of the checksum and the 16-bit `tot_len` field. Use the function `htons` to convert a 16-bit number from host order to network order and `htonl` to convert a 32-bit number from host order to network order.

Use `tcpdump` to create pcap files and `wireshark` to analyze them. Sometimes errors are indicated by wireshark's analysis of the packets; for example, it outputs relative sequence numbers (starting from 0) but the real sequence number is what needs to be used. As well, a number of common errors and tips are listed at the end of this question. The virtual machine automatically runs tcpdump for you if you specify the option when starting it.

The following resources will be helpful:

- `https://tools.ietf.org/html/rfc793#page-15` TCP header format
- `https://tools.ietf.org/html/rfc791#page-11` IP header format
- `man 7 raw` manual for raw sockets

Submit your pcap files, answers to the questions, and the code you wrote to a D2L dropbox that will be set up for this assignment.

Opening Raw Sockets A raw socket allows the person writing to it to place arbitrary data instead of having headers filled out by the operating system. A raw socket also allows the person reading from it to read all network traffic on the system instead of just the data meant for them. Because a single user on a multi-user machine can perform significant attacks on other's network communications, the use of raw sockets is prohibited except for those with superuser privileges.

A raw socket is opened as follows:

```
int s = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
```

You then create a buffer of data:

`char datagram[4096]`
Normally you just send the data, but now the datagram has (in order):

- IP header
- TCP header
- actual data

You then send the packet on the raw socket. The `send` function is only used for connected sockets; raw sockets are not connected. As such, you use `sendto` function and you must include a `struct sockaddr*` for the IP that you want to send to the packet to. Read the manual page for `sendto`: `man sendto`.

Organization The attack code and the client run in the virtual machine. The server runs on the host machine (i.e., linuxlab machine). The attack code that opens the raw sockets must run as root, so use `sudo` to run the command.

To simplify your tasks, it is recommended to disable TCP's optional timestamps. If present the server will reject injected traffic that does not contain them, and the checksum code given below assumes no options in the IP header. You can do this by running the following command on the tinycore VM: `sudo sysctl -w net.ipv4.tcp_timestamps=0`

Attack 1: Fake SYN (16 marks) Recall the client / server code from assignment 3 with the required modifications: (i) the server listens on port 3XXXX, where XXXX is the last four digits of your UCID and (ii) the client sends data to the server. You will write a raw socket program to generate a SYN packet. That is, your program sends a single spoof SYN packet to the listening server and exits.

Remember to run the server on port 3XXXX, where XXXX is the last four digits of your UCID.

Obtain the pcap file from the attack. If the pcap file is rather large or you want to have it less noisy, you can remove ssh communication that may be present: `tcpdump -r fullcapture.pcap 'not port 22' -w filtered.pcap`

Answer the following questions in your report based *on the pcap file that you submit*:

1. Which packet in the pcap file is the spoofed SYN? (The "no." column in wireshark).
2. What is the source port for the fake SYN?
3. What is the spoofed seq number?
4. What is the server's seq number in reply?
5. Does the virtual machine respond to the server's reply?

Attack 2: TCP reset attack (16 marks) Modify the client from the previous attack to sleep for some amount of time before actually sending data, like a few minutes. This sleep time is to allow you a chance to run your attack code before the data transfer begins, which you need to look at the pcap file to get the sequence numbers, acknowledgement number, ports, etc. We'll use this time while it sleeps to reset the connection, thereby effecting a denial of service attack. **Remember to run the server on port 3XXXX where XXXX is the last four digits of your UCID.**

Have the real client program connect to the server and then inject a fake reset message to *only the server* before the real client sends its data.

Obtain the pcap file from the attack, remove ssh as desired, and submit the pcap file and the answers to the following questions:

1. Which packet in the pcap file is the legitimate SYN for the connection that is later RST? (The "no." column in wireshark).
2. Which packet in the pcap file is the spoofed RST? (The "no." column in wireshark).
3. Does the server respond to the spoofed message?
4. Which packet in the pcap file is the legitimate client's later attempt to send data?
5. Does the TCP server that receives the spoofed RST implement the hardening approach described in rfc5961? How did you test for this?

Attack 3: TCP injection attack (16 marks) Use the modified client and server from the previous attacks, i.e., that one that sleeps for some amount of time before actually sending data, we'll now perform a TCP injection attack. During the delay you've added before the client sends its data, we'll inject a fake reply before the client sends its real one. The delay allows you time to get the sequence numbers, acknowledgement numbers, and source ports from the connection to use when spoofing.

Insert whatever data that you want as your reply. Run the real client and the server until the real client attempts to send its data.

Obtain the pcap file from the attack and remove ssh traffic as desired. Submit the pcap and the answers to the following questions:

1. Which packet in the pcap file is the legitmate SYN? (the packet no. column in wireshark)?
2. Which packet in the pcap file did your program spoof (the packet no. column in wireshark)?
3. What is the payload (in hex, so "hello" would be 68656c6c6f) of the spoofed data?
4. How does the server respond to the spoofed packet?
5. Does the TCP server that receives the injected traffic implement the hardening approach in rfc5961 that requires a valid ACK for messages attempting to add data? What did you do to determine this?

Remember for all attacks to run the server on port 3XXXX, where XXXX is the last four digits of your UCID.

Get your code working, then make it configurable You'll have a much easier time doing this assignment if you don't have to recompile your program everytime you change a field. Pass in things like ports, IPs, sequence numbers, acknowledgements, payloads, etc., as arguments. Don't hardcode things: compute the actual packet length and pass it in to the checksum function—otherwise the checksum code that works for attack 1 and 2 won't work for 3. If you have a constant, define it only once and re-use it. Getting these attacks to work can take some time and involve many changes to the code; the last thing you want to spend your time debugging is having the same constant value appearing twice and you only changed it in one place.

To summarize: get your code working, then make it configurable. If you do this and get attack 1 to work, attacks 2 and 3 will be much easier. These exercises build on each other. After you get one to work, it will be useful to modularize the reusable parts to make the next exercises easier. If simple changes involve error-prone code editing and recompiling, you risk reintroducing bugs you have already solved.

Editcap and Tshark After you get your attack working and you have the pcap file for submission, copy the pcap file that is generated to a safe place and analyze it. Since this will have *all* the packets seen in the VM, remove most of the irrelevant ones before answering the questions for submitting. This is done with the `editcap` command:

```
editcap -A "2018-10-28 15:12:00" <infile.pcap> <outfile.pcap>
```

This creates a file outfile.pcap consisting of only the packets after (-A) the specified time. Note that you do not need this to coincide with exactly the first packet for the attack, but use this to roughly cut down the size of pcap files to help in your analysis and submission.

Another approach is to select the packet numbers directly. Suppose you wanted to include packet number 10–40 (i.e., packet no. column in wireshark). This is done as follows:

```
editcap -r <infile.pcap> <outfile.pcap> 10-40
```

Here the `-r` flag indicates *reverse* mode, because the default behaviour for editcap is to remove the selected packets. Do not remove your input pcap file until you are certain that all the necessary information is present in the output pcap file.

Tshark is another tool that can be used for reducing pcap files. Tshark is useful because it shares the filter syntax with the Wireshark GUI. Thus you can use Wireshark to figure out a good filter (e.g., `tcp.port == 3XXXX`) and then use tshark to filter your capture file. This is done as follows:

```
tshark -r <infile.pcap> -w <outfile.pcapng> -Y "<wireshark filter string>"
```

Tips and Caveats Raw socket programming is hard because if anything is wrong with your packet it will be ignored or sent somewhere else on the Internet. This makes debugging hard. Many things can go wrong and it's hard to get feedback about why, and if any one thing is going wrong then its just not going to work. What follows is a set of tips, reminders, things to do to avoid too much frustration, etc.

- Use command line parameters for values that you find yourself tinkering or changing frequently instead of changing the code and recompiling it. Use `atoi` to turn an `argv` value into an integer.

- Check all your return values for errors. C functions often return an error value and set `errno` if an error occurred. Error numbers are not human meaningful but `strerror()` returns a string description of it. Errors should not normally occur.

- You need to either be root or have `CAP_NET_RAW` to open raw sockets. You can give a program the capability as follows: `sudo setcap cap_net_raw+ep <file>` If you ever find yourself running such code on your own machine for some reason, do not run it as root but grant the raw socket privilege. (What security principle is this?)

- Use wireshark and the valid client-server running to see what valid, operating-system produced packets actually look like. If you see your packets on wireshark but they are being ignored or not being responded to, then try to make them look closer to what you can see actually works from the operating system. Make sure you understand what the header fields mean as you change them and why it makes sense to change it. Don't just make changes.

- Don't send your traffic to or from localhost (127.0.0.1) on the tinycore machines. We are monitoring its real network interface, so use its IP. (127.0.0.1 is actually a "loopback" network separate from the Internet.) Use `ifconfig` to check its IP. Note that its IP may change when you restart it! Run the server on the host and the client on the VM.

- Wireshark uses *relative* acks and syns, meaning that no matter what the syn is used, it prints the first one as 0 and it goes from there. Look at the raw packet to see the bytes, because the actual syn number is different.

- Don't forget about host-order and network-order endianness!

- Seriously! Endianness matters and it affects all numbers longer than a byte for any context in a network packet.

- You must compute checksums for headers. Learn how they are exactly computed and implement that. Check you checksum computing routine by taking actual packets from pcap and running your code on them! The checksum you get should be the same as the one actually being sent. Note that you must specifically clear the checksum fields in such examples (i.e., make them zero) before computing the checksum over it. Code to do this appears below in the assignmnet.

- Use constants and variables for things. Don't hardcode values like IP, ports, packet lengths, etc. Values like this that can either change or be needed in multiple places are an easy way to waste time debugging something you'll regret when you find out that you changed it in once place and forget to change it in another.

- Use structures for things like the TCP and IP headers instead of just settings bits on a big `uint8_t *` buffer and sending it over the network. There are already nice ones defined. For example, the header files <netinet/in.h>, <netinet/ip.h>, and <netinet/tcp.h> are quite helpful as they define `struct iphdr` and `struct tcphdr` along with constants for flags already for you. You can find these files in `/usr/include` on most systems (e.g., `/usr/include/netinet/in.h`). These allow you to change fields

reliably and easily. If you are writing code like `memcpy(buf+46, val, 8)` you will find it easier if you code looks more like like `memcpy((void*) &(tcp_hdr->field), val, sizeof(val))`. The tutorial raw socket code helps with this.

- Use version control or copy code that is working. If you keep changing the same file you may cause something that use to work to stop and it is hard to go back without good version control.
- Remember to disable TCP timestamps on the server machine.
- Duplicate ACKs are a signal that your packet is malformed. It indicates that the packet was received by the host machine but rejected by its TCP stack (e.g., due to a bad sequence number).
- Print out any arguments you pass into your program. Make sure you convert strings to integers correctly (e.g., use `atoll` for sequence numbers).
- Sometimes it can take some time for the TCP handshake to appear in the pcap file. Make sure you are looking at the right handshake. Filter the pcap for only traffic involving the server's TCP port and the last three lines should be the handshake (assuming you are setting the port correctly).

Checksum Code This function computes a TCP checksum. Pass in a pointer to the start of the packet (including IP header) casted to an unsigned short pointer, and the number of two-byte words in the TCP payload (not head). Thus, sending 4 bytes means 2 words. Note that this code assumes a fixed length TCP header of 10 words (20 bytes), so optional things like TCP timestamps are disabled.

```
unsigned short csum_tcp(unsigned short *buf, int nwords) {
        unsigned long sum = 0;
        sum += buf[6];
        sum += buf[7];
        sum += buf[8];
        sum += buf[9];
        sum += htons(6);  // 6=TCP protocol in IP header
        sum += htons(20 + (nwords << 1));  // length of TCP header and payload
        for (int i = 10; i < 20 + nwords; ++i) {
                sum += buf[i];
        }
        while (sum >> 16) sum = (sum >> 16) + (sum & 0xffff);
        return ~sum;
}
```

This function computes the IP header. Pass in a pointer to the beginning of the packet casted to an unsigned short pointer, and the number of words for the IP header only (this will likely be 10).

```
unsigned short csum_ip(unsigned short *buf, int nwords) {
        unsigned long sum;
        for (sum = 0; nwords > 0; nwords--)
                sum += *buf++;
        while (sum >> 16) sum = (sum >> 16) + (sum & 0xffff);
        return ~sum;
}
```

To check your implementation you can use the following packet:

```
RRAATGqTQABABgAACgACDwoAAgIAFqC8wY0dNQATvdtQGJkgAAA
AAAAAABC/auG+hRBfKXpfGgJ6rcBt2F4JWfijNYP+4uFFcX5IEA==
```

This is base64-encoded because it is binary. Base64 decode it to a file, and check the checksum. This file starts at the IP header, and has the IP and TCP checksum fields zeroed out. If you compute the checksum with the correct algorithm, you'll get 0xb7f8 for IP and 0xc26c for TCP checksum. Be sure to consider endianness when putting the checksum into the corresponding fields.