Question 1: TCP Injection Attacks (40 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 will need to be able to inject raw packets and monitor a network interface. This requires root access. To help you with this, we have setup a virtual machine that you can use on laboratory computers. You will have root access on these machines. Details on how to use the virtual machine is listed at the end of this question. The submission for the exercises are (i) your C code, (ii) a packet capture file ('pcap') that exhibits your attack in action, as well as a (iii) document that goes with the pcap file and references it. The document will answer questions and, for example, identify which packets in the pcap file corresponds to your attack.

Sending raw packets is tricky and there is often little indications of why it isn't working. Bad packets are just rejected by the operating system. You have to set all the TCP and IP header values correctly. Use `tcpdump` to create pcap files and `wireshark` to analyze them. Sometimes errors are indicated by wireshark's analysis of the packets. 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

Your code must be written in C and you may not use external libraries. For all questions, submit the pcap file, your C code, and a report answering questions and referencing the pcap file.

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 will need to store the following (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`.

Attack 1: Eavesdropping (4 marks) Recall the client / server code from the first tutorial. Compile and run the server so that it is listening on the port: 3XXXX, where XXXX is the last four digits of your UCID.

Run the server and the client while monitoring the traffic. Obtain the pcap file and answer the following questions:

1. What are the number of the three packets for the TCP handshake. (The "no." column in wireshark).
2. What is the server's sequence number
3. What is the client's sequence number

Submit the pcap file, your C code, and the answer to these questions.

Attack 2: Fake SYN (10 marks) Recall the client / server code from the first tutorial. Compile and run the server so that it is listening on the port: 3XXXX, where XXXX is the last four digits of your UCID. 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.

Obtain the pcap file from the attack and answer the following questions:

1. Which packet in the pcap file did your program spoof? (The "no." column in wireshark).
2. From which IP and port did you send the fake SYN?
3. How did the server respond? What happens after that? Describe the packets that follow in the pcap file.
4. What is the spoofed seq number?
5. What is the server's seq number?
6. If you run your program again right after, does the server consider it a new connection attempt or does it not? Explain why it is the case. If it thinks it's a new connection, explain how you could change your attack code to make it not think so; if it things it's not a new connection, explain how you could change your attack code make it think so.

Submit the pcap file, your C code, and the answer to these questions.

Attack 3: TCP reset attack (12 marks) Modify the server from the previous attack to sleep for some amount of time before actually responding to the client. This sleep time is to allow you a chance to run your attack code before the data transfer begins. 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 client connect to the server and then inject a fake reset message to either party.

Obtain the pcap file from the attack and answer the following questions:

1. Which party do you send the reset message to?
2. Is this sufficient to disrupt communication?
3. Which packet in the pcap file did your program spoof?
4. Which IP and port did the client connect from?
5. What happens to the running client after the spoofed reset is sent?

Submit the pcap file, your C code, and the answer to these questions.

Attack 4: TCP injection attack (14 marks) Use the server from the previous attack, i.e., that one that sleeps for some amount of time before actually responding to the client and runs on port 3XXXX where XXXX is the last four digits of your UCID. We'll use this timeout to send spoofed data.

Have the client connect to the server and then inject a fake reply before the server sends its reply. Insert whatever data that you want as your reply. The client should behave no differently than if the server had actually sent this data.

Obtain the pcap file from the attack and answer the following questions:

1. Which packet in the pcap file did your program spoof?
2. Which IP and port did the client connect from?
3. What happens after the spoofed data is sent?

Submit the pcap file, your C code, and the answer to these questions.

Virtual Machine The virtual machine is accessible from the undergrad computing environment.

`/usr/local/tinycore/tinycore -f <snapshot file> -c <capture file>`

The capture will grab data leaving and entering the VM. Running the server on the VM is also hard to reach. Therefore *run the server on the host machine, run the client on the VM, and run the attack code on the VM.*

The snapshot file is used to store the state of the VM and the capture file stores the live output of `tcpdump` as the VM runs. As packets go through the VM it is recorded to this file, which can be examined using `wireshark`.

You can `ssh` into the VM by following the instructions. It should be `ssh tc@localhost -p <VM's ssh port>` and the password should be `CPSC526Pass`. The port is dynamically assigned and printed to the console when it starts. If you are logging in remotely to the lab machines you should use `ssh -X` to allow X-forwarding for the VM's GUI. Note that you can do all of the steps for this work using the console, so the laggy GUI on the VM is not a concern: simply ssh into the actual VM and run your programs like that.

Your account should automatically allow you to run commands prefixed by `sudo` to grant super user privileges. Note that the use of ssh and similar commands generates network traffic which is recorded in the pcap file. Wireshark supports "filters" to remove this easily identifiable traffic to help you see what is important. *It is recommended to spend some time learning how to use wireshark's features.*

The VM may not `gcc` available to compile your programs. To install it, run the following from *inside* the VM: `tce-load -wi compiletc`

Because of quirks of networking, you cannot run the server and the client both on the VM and correctly monitor the traffic. Consequently, run the server on the host machine and the client on the VM. Thus, the program running on the VM connects to the server which is running on IP of the machine that launched the VM.

Don't save data on the VM Do not work on the files inside the VM as no guarantees are made on its storage. Work on your files in your normal lab environment, and copy them to the VM using `scp`:

`scp -P <VM's ssh port> <filename> tc@localhost:`

or

`scp -P <VM's ssh port> -r <directory> tc@localhost:`

Repeating: **do not store your work inside the VM and expect it to remain available**. If you shutdown the VM using the GUI screen your data should persist. Maybe. Who knows. **Do not rely on this**. Don't store data on the VM.

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 as arguments. 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.

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.

VM SSH Key Change It can happen that you see the TOFU-error for SSH:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
```

This happens whenever the public key for the VM changes (which can happen if it gets reset). This is how ssh uses public keys. When you first connect you see:

```
ECDSA key fingerprint is SHA256:p7tU6g/OhEMUB8Py6c3Od7OMaO4Ktz/1Eijq37aaVNE.
Are you sure you want to continue connecting (yes/no)?
```

Typing yes means you either (i) verified that this is the correct key, or, more likely, (ii) use TOFU as your security model. The all-caps warning is the warning when the TOFU-model detects that the key is changed, indicating a possible man-in-the-middle attack. In this case, it is just that the VM reset and has a new key. After the warning it includes the instructions to remove it:

```
Offending ECDSA key in /home/<your home dir>/.ssh/known_hosts:X
```

where X is a line number. If you remove that line from that file you can log in again.

Editcap 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.

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 to 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 <file>` If you are running this on your own machine, you should not run it as root but rather with this raw packet capabilities.
- 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.
- 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))`.