Question 1: TLS Downgrade (10 points)

In class we discussed how the TLS protocol allows the browser and the server to agree on a cipher suite, which can include a key exchange algorithm, authentication algorithm, encryption algorithm, and a MAC algorithm.

Part 1: How do the browser and the server decide on which cipher suite to use? Who ultimately controls that decision?

Part 2: Suppose that Eve knows how to break some set of cryptographic primitives, that is, Eve knows how to break some elements of a cipher suite, but not *necessarily* all elements for a particular cipher suite. For example, Eve knows how to break only 128-bit AES, but neither 256-bit AES nor any other algorithm in a cipher suite. Suppose that both Alice and Bob.com support a ciphersuite that includes the algorithm that Eve can break. Eve can use this to mount a man-in-the-middle. What are the steps in this process? What tries to prevent this and which elements of the cipher suite does Eve need to be able to break in order for the attack to work? What can Alice and Bob.com do to protect themselves?

Question 2: TCP Injection Attacks (30 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. You can also use your own machines if you find that more convenient. The submission for the exercises is a packet capture file ('pcap') that exhibits your attack in action, as well as a 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. You have to set all the TCP and IP header values correctly. Use `tcpdump` to create pcap files and `wireshark` to analyse 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 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

Opening Raw Sockets A raw socket allows the person writing to it to place arbitrary data instead of having headers filled out. 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: Fake SYN (10 marks) Recall the client / server code from homework 1. 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. Which IP and port did you send the inital SYN from?
3. How did the server respond? What happens after that?
4. What is the spoofed seq number?
5. What is the server's seq number?

Bonus (2 marks): Implement an actual SYN-flood attack on the server. Does it succeed in stopping service? Obtain the pcap file and find the first packet that isn't replied to.

Attack 2: TCP reset attack (10 marks) Modify the server from the previous attack to sleep for some amount of time before actually responding to the client. 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?

Bonus (2 marks): Use raw sockets to also *read* local traffic from the wire, and perform this attack automatically, as soon as the connection is established. Remove the timeout from the server. Run the attack 10 times. Provide the pcap file and the answer to the question: How often does your reset packet arrive before the actual server data.

Attack 3: TCP injection attack (10 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?

Bonus (2 marks): Use raw sockets to also *read* local traffic from the wire, and perform this attack automatically, as soon as the connection is established. Remove the timeout from the server. Run the attack 10 times. Provide the pcap file and the answer to the question: How often does your spoofed data get received before the actual server data.

Virtual Machine The virtual machine is accessible from the undergrad computing environment. Start it by typing:

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

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 should be 5555 but it is dynamically assigned and depends on the load and the machine. Your account should automatically allow you to run commands prefixed by `sudo` to grant super user privileges. Note that the use of ssh 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.

The VM should have `gcc` available to compile your programs. Do not work on the files inside the VM as no guarantees are made on its storage. Work on your files in your normal environment, and secure copy them to the VM:

`scp -P <VM's ssh port> <filename> 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. But **do not rely on this**.

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/OhEMUB8Py6c3Od70MaO4Ktz/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.

After you get your attack working and you have the pcap file for submission, copy the pcap file from the VM to a safe place and analyse 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. You do not need this to coincide with exactly the first packet for the attack, but only to avoid having enormous pcap files to analyse and submit.

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. Alot of things can go wrong and it's hard to get feedback about why. What follows is a set of tips, reminders, things to do to avoid frustration, etc.

- Check all your return values for errors. C functions often return an error value and set `errno` if an error occured. Error numbers are not human meaningful but `strerror()` returns a string description of it.
- 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>`
- Use wireshark and the valid client-server running to see what the packets should look like. If you see your packets on wireshark but they are being ignored, try to make them look closer to what isn't being ignored—but 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 localhost or 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!
- If your packets aren't appearing in wireshark it may be because they are not having the VM as either its source or destination. Running the server in the VM makes sure that one of the two fields will always be monitored.
- 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!
- 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 alot of 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 data 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`)