Consider a garage door attached to a house. The door cannot be opened aside from a remote control in a vehicle, and the door from the garage to the house is thus often left unlocked. In the context of a weakest link, opening the garage door may very well be the easiest way to enter a house. A traditional way that a garage door opener works is to emit a secret that opens the door.

This assignment consists of a number of different versions of the garage-door opening protocol, each of which is vulnerable to an active attacker. Your job is to observe successful protocols, and then perform an attack. You'll perform eavesdropping, replay, and brute-force attacks in order to break the security. A successful attack will have the garage door respond with `SECURITY BREACH: DOOR OPEN! code: XXXXXX`, where `XXXXXX` is a hexstring like `18A7874F` that you submit to show you have attacked the door successfully.

You can submit the list of these codes on D2L along with a paragraph description of what you did to break into the system.

Basic Idea

The basic protocol is as follows:

1. create a TCP connection to the garage server
2. issue a `connect` command to start the session
3. `issue commands` to the door
    - open door
    - close door
    - turn on lights
    - turn off lights
4. issue a `disconnect` command to end the session
5. close the TCP connection

Each version of garage door has a different protocol, for example, by encrypting some messages, or by having the server reply with a challenge. A basic description of the protocol is provided for each version in this assignment.

Each version of the garage door comes with two programs: `garage_server` and `garage_observer`. The `garage_server` program takes as command line argument the last four digits of your UCID. For example, if your UCID were 12345678 you would run it as `./garage_server_v1 5678`. The `garage_observer` programs takes two parameters: the IP address and last four digits of the UCID.

To perform this assignment, run the observer on the VM and the server on the host. Get the IP of the host with `ifconfig` to tell the observer to connect to it. Use the last four digits of your UCID for both the server and the host, and it will use a deterministic port that will hopefully not collide with other things running. Each version uses a different port based on your UCID, so you don't need to worry about accidentally running version 2 of the server and version 3 of the observer.

The observer connects to the IP and port of the garage server, and issues a few commands based on the protocol. It outputs what you observe and the timestamp of that observation, such as the lights coming on. Thus, `garage_observer` simulates someone else having real-world interaction with the door while you are observing what is happening. When running the `garage_observer`, you should record the network traffic using `tcpdump`; tinycore supports this as a startup option so you have a record of the network traffic. Note that it can only monitor traffic that leaves the VM, hence why we run the server and observer on separate machines.

You can copy the `pcap` file that is being generated to your local machine and view it with wireshark, or directly if you are not SSHing into the host machine running tinycore. Note that this pcap file will also include your interactions with the VM over SSH. To remove those, apply the filter `tcp.port != 22` in wireshark, or create a new pcap with those removed by running tcpdump:

```
tcpdump -r fullcapture.pcap 'not port 22' -w filtered.pcap
```

Moreover, in wireshark you can right click a relevant request and click "Follow" and "TCP stream" to see the exact back and forth communication of TCP payload data. If you find one of the packets relevant to the exercise, this allows you to see the full conversation.

Wireshark may display things using time relative to the start of the capture. To replace with absolute time, you can right click the time column header, go edit column, and change the display type to absolute time. This allows you to readily match the timestamps of observed events (e.g., garage door opened) to contemporaeous network traffic in the pcap file. Note that there may be some small different in time (i.e., less than a second), as well as a round number of hours of difference. You'll get a sense for the clock skew in the first exercise where there is no encryption so mapping commands to effects is straightforward.

Once you have figured out how to attack the system, you can connect directly to the IP and port and send the relevant traffic. Be sure to read the reply to get the code if you are successful. The goal for each version is to do the following:

1. connect via TCP to the garage server
2. issue a connect command
3. issue an open door command
4. issue a disconnect command
5. close the TCP connection

.

It will become clear how to issue these commands with the first protocol, because it is not encrypted. Capture the network traffic and relate it to what occurs to figure out how to issue the relevant commands. You are free to run the observer program as many times as you want, and it may be necessary to do so because its behaviour changes across different runs.

In addition, a program `garage_attacker` is also provided. This is to help mount the various attacks. It takes as its arguments the IP and *port* (not last four UCID) for the garage server. Remember that each different version runs on a different port, so look at the output of `garage_server` to see the port on which it is running. The `garage_attacker` program then reads a line of input encoded in hex and sends the actual bytes of that input over the socket. This is to help with the later versions of the protocol where the data transmitted is encrypted. If you want to replay a message from the captured network traffic, you can select a field in `wireshark` and choose *copy* and from the submenu *as a hex stream.* The bytes that are copied are formatted how `garage_attacker` expects and will effectively resend that data. You can also pipe in a file or output of another program, so if you create a list of data you want to send in a file `attack_traffic` you can type `cat attack_traffic | ./garage_attacker (IP) (port)` to send the file as commands line by line.

You may find it useful to turn off the VM after each exercise, delete the pcap file and restart it, so that network traffic from previous exercises are not present during your analysis. This is not necessary, however, because each exercise uses its own particular port; be sure you are looking at the right traffic.

Moreover, you may find it useful to try to think of how you will attack the protocol first, by reading the description. If you do not see an obvious attack, then try running it and looking at the network traffic to see if that gives any ideas. Remember some of the key concepts we covered in class: replay attacks, cryptographic malleability, known plaintext (i.e., defenders have protocols), two-time pads, and what information is leaked even when something is semantically secure. Rule out any that clearly do not apply, and then try to figure out how the remaining ones may apply.

Running the Code

The binaries for this exercise and a script for the tinycore VM to capture network traffic are available in this tarball:

`https://cspages.ucalgary.ca/~joel.reardon/526/cpsc526_garage.tar`

These are to be run *on the cslinux undergrad environment.* Instructions for how to do this are in here: https://cspages.ucalgary.ca/~joel.reardon/526/hw/hw2.pdf that is, the third tutorial session called *tinycore*. You can SSH into the cslinux environment to do this from home, or you can use a lab machine directly. Make sure you do this tutorial session first before beginning on this exercise.

Be sure to run the software as follows:

- Run the `tinycore` VM on a cslinux host machine.
- Run the `garage_server` program on the same cslinux host machine.
- Run the `garage_observer` program *in the tinycore VM.*

This will allow you to correctly capture the network traffic between the observer and the server as it goes from the VM to the host. If you run both programs on the VM or the host, you will not be able to see any relevant network traffic.

Instructions to download the code:

1. Download the tarball to the cslinux environment:

   ```
   wget https://cspages.ucalgary.ca/~joel.reardon/526/cpsc526_garage.tar
   ```

2. Extract the files:

   ```
   tar xvf cpsc526_garage.tar
   ```

   This gives you three directories:

   - bins_for_host: holds the binaries to run on the host machine, i.e., the cslinux machine.
   - bins_for_vm: holds the binaries to run on the tinycore VM.
   - vm: script to launch tinycore.

3. Launch the VM as per tutorial exercise. You can either `scp` the bin_for_vm directly, or do the `wget` and `tar` process again inside the VM.

A tutorial session will be devoted to this assignment to help make sure you are running it all correctly.

Rememeber that you *do not need to use the GUI of tinycore at all.* You are welcome to if you want, but all of the exercises can be done on the command line. You can *ssh* into the tinycore VM on the command line instead of starting a terminal sessions inside the VM, and this will have far less lag.

Version 1 (8 points)

The first version of the protocol does not use encryption. Your job is to figure out basic protocol, e.g., how to connect, open doors, etc., and then issue your own command by creating your own client. The remaining versions of the system have the same underlying protocol, but with encryption, tokens, or other fields added.

Version 2 (8 points)

Version two is improved by using encryption! Unfortunately, the developers used AES in ECB mode (oops). There is a shared key between the garage door opener and the garage, and all messages are encrypted with 128-bit AES in ECB mode with that key. You do not need to figure out the key, however, in order to figure out how to leave the door opened.

Version 3 (8 points)

Version three is improved by using 128-bit AES in CBC mode (much better). Better still, they actually use random per-message IVs. Each message that is sent from the client to the server now has the following structure:

```
+------------+------------+
| 16-byte IV | E(message) |
+------------+------------+
```

Thus, a message like `open` will now be sent as a 16-byte IV followed by the encryption of the message using the shared key and the sent IV. Again, you do not need to learn the key in order to attack the system.

Version 4 (8 points)

Version four is the same as version three, except that AES is used in counter mode (CTR). That is, it functions like a stream cipher and not a block cipher. The IV used continues to be send prefixed to the encrypted message, as with version 3. The corresponding observer program, however, no longer reports what is happening when the garage door is being used. That is, commands will be issued and will appear in the packet capture, but you won't be able to relate the packets to what happens to the garage. What information gets leaked by a stream cipher as compared to a block cipher?

Version 5 (8 points)

Version five is the same as version four except that a two-byte random challenge is provided by the server to the client as a reply to the connect command and all subsequent commands must include this challenge as a prefix to the command, and it is encrypted as well. Thus, the protocol looks like the following:

```
client connects to server:
+------------+--------------------+
| 16-byte IV | E(connect message) |
+------------+--------------------+

server response to connect:
+------------------+------------------+
| 2-byte challenge | connect response |
+------------------+------------------+

client sending non-connect commands:
+------------+--------------------------------+
| 16-byte IV | E(2-byte challenge | message) |
+------------+--------------------------------+
```

Each time the connection happens, the server gives a new random two-byte challenge to the client. If you run the observer, it is likely that the challenge that you capture won't be the same as the challenge that you are given were you to replay the message. Nevertheless, the fact that it is a stream cipher allows for a malleability-based attack to break into the system.

When doing this you may find it useful to take the hex commands from wireshark and copy them to a file where you can add the spaces, so when making modification you know you are modifying the correct components.

Version 6 (8 points)

Version six is a modification of version four except that instead of using a random IV, there is a fixed IV of all zeros. That is, every message is encrypted with the same key and IV using AES in counter mode (a stream cipher). The catch is that during the observer, you won't see any garage door opening commands, i.e., you can keep running it but an open will never happen. You'll have to figure out how to craft such a message given. Hint: the use of counter mode with IV reuse results in what problem?

Because the IV is fixed, it is no longer transmitted. Thus the client to server messages have the following structure:

```
+------------+
| E(message) |
+------------+
```

Version 7 (0 points)

This is not worth any points, but available to be done if you are interested. It is the same as version five, but instead of a 2-byte challenge, it is a 4-byte timestamp.

Version 8 (0 points)

This is also not worth any points. It is the same as version five, but instead of a 2-byte challenge, it is a 16-byte random value. It is not securely generated randomness, but instead generated from this code:

```
string challenge;
for (int i = 0; i < 16; ++i) {
        char c = (char) rand();
        challenge += c;
}
```

Hint: how is `rand()` typically seeded?

Virtual Machine Instructions

Refer to tutorial handout on tinycore for further instructions. Your TAs in tutorial will help with making sure you are running this exercise correctly.