Name: _____

IB Computer Science 2
Presentation-layer protocols worksheet

Presentation-layer protocols sit one level of abstraction below application-layer protocols, meaning that they carry an application-layer protocol as their payload.

Recall that in an application-layer protocol such as HTTP, packets consist of a header (containing protocol metadata such as the domain name and file path) followed by the data. A presentation-layer protocol is the same, except that the data is the application-layer packet.

One common presentation-layer protocol is Transport Layer Security (TLS), which carries application-layer packets in encrypted form. (Despite the confusing name, TLS is not a transport-layer protocol!) When TLS carries HTTP data, the two are referred to as HTTPS.

Without HTTPS, you would have to trust every network between you and any Web server that you visited. If any of these networks contained a user who could observe your HTTP packets as they went by, that user could read the requests you were making and the files the server was responding with. Not good if you were providing a means of payment or checking your grades!

1. Open a terminal and start the command `nc -l http`. Enter `localhost/file` into your Web browser's location bar and look at the output in your terminal. If someone snooped on this request, list at least two things they would learn.

2. Press Ctrl+D to exit `nc`, then rerun it with `nc -l https`. Navigate your browser to `https://localhost/file`. What happens in your terminal and why?

3. Recheck the output. Notice how there is some unencrypted data at the beginning: this is the TLS header. If someone snooped on the request, how much would they learn?

4. Conduct a Web search and look at the location bar. If someone snooped on your browser's request, would they know what you were searching for? Why or why not?

Making sense of TLS-protected data requires both the client and the server to have secret encryption keys. Press Ctrl+D, then generate a server key by running the following command:
`openssl req -new -nodes -x509 -keyout server.pem -out server.pem`
Enter your two-letter country code but leave the other prompts blank.

5. Start the command `openssl s_server -accept https`. Return to `https://localhost/file` in your browser. In the resulting error page, click the Advanced or Show Details button to read more about the error. What does your browser not recognize as valid or trusted?

6. A server's certificate serves as proof that you are really connected to the Web server for the domain name that you entered. However, even without validating the certificate, you would still be able to establish a secure, encrypted channel with the server. Why would this not be enough to keep your information secure?
   Hint: It may help to think back to captive portals. How could an attacker use a similar technique against you?

7. Click the link to continue to, proceed to, or visit the site despite the problem. What do you see in your terminal?

8. Try responding with the following:
   ```
   HTTP/1.1 200 OK
   Content-Length: 11

   Hello world
   ```
   Hit enter at the end. What happens?

If you have extra time…

With your `openssl` server still running, open a new terminal and type `hostname`.

On another computer, run the following command, replacing `hostname` with what you saw:
`openssl s_client -connect hostname:https`

Switch back to the `openssl` server terminal. You can now securely "chat" between computers!

Name: _____

To encrypt data before sending it to the server, the browser needs a key. You may be wondering how the server can avoid sending the client its own secret key before the encrypted session starts, since if anyone snooped on this message, they would be able to decrypt everything!

To initiate the connection, the server uses a special kind of encryption called public-key cryptography. There is actually a pair of keys, a public key that is used to encrypt messages and a corresponding private key that is used to decrypt them again.

Earlier, you generated the private key file `server.pem`. Derive its corresponding public key with the command: `openssl rsa -pubout -in server.pem -out client.pem`

9. Open both `server.pem` and `client.pem`. Ignoring the certificate, what do you notice?

10. Run the following command:
    `openssl rsautl -encrypt -pubin -inkey client.pem -out secret.txt`
    Enter a secret message of your choice, then hit enter followed by Ctrl+D. What do you find in the `secret.txt` file?

11. Run the following command:
    `openssl rsautl -decrypt -inkey server.pem -in secret.txt`
    What do you see?

12. Run the following command:
    `openssl rsautl -decrypt -pubin -inkey client.pem -in secret.txt`
    What happens and why?

13. Based on your earlier observation about the key files, do you think it is easy to derive the private key from the public key? Why or why not?

14. The command `openssl rsa -text -in server.pem` shows the private key's components. Compare against `openssl rsa -text -pubin -in client.pem`. Can the server safely send the public key to the browser unencrypted? Why or why not?