Final Project Writeup

Project Description

For this project Sam Breese and I decided to implement TLSv1.2 (RFC found at https://tools.ietf.org/html/rfc5246) with the eventual goal of being able to connect to Google. Unfortunately, we did not have time implement several elements that are required for "proper" TLS, but we should still retain most of the security. The actual application is a simple echo server and client. When running the program, the user selects the port, hostname, and whether to act as the server or client. The server listens for connections from clients, which it then handshakes with and waits for messages, which it decrypts, reverses, then sends back re-encrypted. The client connects to the specified server, handshakes, then waits for command line input which it encrypts and sends to the server, while also listening for server messages which it decrypts and prints to the command line. This is a really simple setup and serves more as an example use case than as an actual program. Of course, our original idea would have been a much more interesting test, as you could then use it to send arbitrary communications with extant services.

What I did

I mostly worked on the protocol implementation, although a lot of that didn't go into the final project. The library I was using for serialization to ByteStrings that we could pass over the network was a bit wonky. It seemed to arbitrarily pad the ByteStrings with extraneous 0s, which does not meet the spec at all. To get messages in the correct format we had to throw out that library and hand-write new code. Once we did that, I wrote the server-side network code and client message deserialization (the server has to deserialize client messages of course, and vice-versa).

I also wrote a size-typed unsigned integer type using Sam's size-typed vector type. We didn't end up using it but if I were to add proper serialization support it could be hugely helpful in sending messages over the network, you could include it in a structure and it would automatically serialize to the right length (the 16-bit number 10 should not serialize to the same value as the 32-bit number 10, for instance).

An interesting part of this project was that it's really easy to see working examples of our end-goal, for debugging I sometimes opened up WireShark and looked at the actual TLS messages being passed around. This gave me a much greater appreciation for the complexities hidden in just opening up a simple webpage.

Implementation Details

We wrote the project in Haskell using Stack for package management and as a build tool. When we started the project we wrote a large type-level interface for ciphers and hashes. Unfortunately we had to scrap most of that so that we could actually get the project done, but I would like to go back to this at some point and get that stuff working. I think it would force us to organize our code in a way that would be a lot easier to understand at a glance – right now our code is several hundred lines of rather obtuse Haskell code, and the client and server network code shares a huge amount of redundancy, while the serialization and deserialization is rather ad-hoc and brittle. Because we only used the one stream cipher there wasn't really a point in making the network code polymorphic in the cipher used, but if we were to add support for more cipher suites it would be invaluable to have structures that allow us to abstract away specific cipher implementation details.

TLS Unimplemented Parts

TLS requires record fragmentation, which acts as a wrapper over the raw TCP connection that TLS uses. The record layer abstracts many of the network realities away from the cipher implementation. Because we did not have the record layer, we sent the raw TLSCiphertext over the network. This works fine while sending small encrypted messages back and forth like in a chat app, but we would have issues handling larger streams of data such as a file transfer.

TLS mandates support for quite a few cipher suites and allows for extension with more "private" cipher suites. Each cipher suite contains the various encryption, decryption, and MAC functions necessary for the body of the communication. We implemented `TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256`, which uses ephemeral ECC Diffie Hellman for the key exchange, RSA for handshake authentication, and ChaCha20 as a stream cipher with built in authentication. This is quite secure, Google has it second in their list of supported cipher suites ranked by preference. We could extend our implementation to support other cipher suites relatively easily, although it would require modifying our key exchange code.

TLS also supports various smaller bits like message compression and optional extensions. Some of these features are negotiated in the handshake, but right now all we do is send out constant data and ignore it on the receiving end. The biggest of these is probably the alert system, which allows the client or server to report things like incorrect MACs or decryption failures. Our system will refuse to continue the conversation but won't report why.

Messages are required to have sequence numbers that increment by one for every message sent. We handle this part fine, but TLS requires that sequence numbers be 64-bit unsigned integers and that they do not wrap around, in other words you can only send a finite number of messages before a second handshake must take place and new keys must be chosen. We do not handle this part and instead the sequence numbers wrap around. This isn't a huge deal for our use case, it's a bit absurd to

be sending 10^{20} messages in a chat program. Even if they do overflow that should only weaken the message authentication, encryption will still be secure.