Mandatory Assignment 2

The protocol - Hash based Commitments over TLS

Hash based Commitments

1. Alice and Bob, decide on
    - a Cryptographic Hash Function H
    - a salt size k and
    - a function $R: n,m \to x$ where n and m is numbers in the range 1-6 and x is a number between 1 and 6.
2. Alice generates a random number m
3. Alice generates a salt s of length $k-1$
4. Alice concatenates s and m to get $s|m$
5. Alice hashes $s|m$ with H to get $H(s|m) = c$
6. Alice sends c to Bob
7. Bob generates a random number n
8. Bob send n to Alice
9. Alice sends s to Bob
10. Bob uses s and the previous message containing m to compute $H(s|m)$ and compares to the previous message c
11. Alice and Bob computes the dice outcome $R(m,n)$

TLS

All the messages from the Hash based Commitments protocol is sent using TLS 1.3.

TLS requires a public key infrastructure afterwhich it

1. does public key exchange
2. to establishes shared secrets for symmetric encryption
3. such that you can, send encrypted authenticated data

Why the protocol is secure

Hash based Commitments

Hashing a number in the range [1;6] can be easily brute-forced, since it only requires creating the 6 different hashes and compare it to the hashed message to know what alice has sent. But prepending a large salt, before hashing introduces an amount of randomness, expanding the input space, and making it infeasible to brute-force.

With this protocol neither Alice nor Bob can "cheat", in the sense that, neither party can reliably produce any outcome they desire.

Alice can't cheat since alice has committed to a number before knowing Bob's dice roll.

Bob can't cheat since Bob, cannot decipher Alice's commitment, and therefore has to send his diceroll to Alice before receiving the information to verify the commitment.

TLS

TLS 1.3 is a protocol that creates a secure channel that ensures, Authenticity, Integrity and Confidentiality.

It achieves confidentiality through symmetric encryption, which is setup through a public key exchange.

It achieves data integrity by using Message Authentication Codes.

For authentication you can opt into two-way authentication, using authentication certificates, or in other words, a public key infrastructure.

The implementation

I've implemented this using two languages C# and javascript.

C#/.NET

The C# part of my codebase is responsible for orchestrating/timing and encoding/decoding messages, as well as hashing messages when needed, for the commitments.

I've chosen the Cryptographic Hash Function H to be System.Security.Cryptography.SHA512::ComputeHash from C#/.NET.

I've chosen the salt size k to be 512, since it matches the hash size of SHA512, a hashing function in the family of SHA-2, which should be difficult to break.

I've chosen $R : n,m \to 1 + (((n-1) + (m-1)) \mod 6)$, since its simple and gives a uniform probability distribution for the range [1;6]

Most of the cryptographic functions are in the shared library/class "Common", where I use the classes `SHA512` and `Random`.

```
namespace Common;
using System.Security.Cryptography;

public static class HashBasedCommitments
{
  private static SHA512 sha = SHA512.Create();
  private static Random rand = new Random();

// [...]
}
```

`GenerateRandomDiceRoll` Generates a random number in range [1;6]

```
public static int GenerateRandomDiceRoll()
{
```

```
    return 1 + ((int)Math.Floor(rand.NextSingle() * 6) % 6);
  }
```

`GenerateSalt` Generates a 512bit byte array of random bytes

```
  public static byte[] GenerateSalt()
  {
    var sizeInBytes = 512/8;
    var buf = new byte[sizeInBytes];
    rand.NextBytes(buf);
    return buf;
  }
```

The results of the two functions above are used in `GenerateCommitmentHash` which concatenates salt and dice, and then hashes to a 512bit byte array

```
    public static byte[] GenerateCommitmentHash(int dice, byte[] salt)
    {
      var l = salt.Length;

      // clone byte array
      var con = new byte[l+1];
      for (int i = 0; i < l; i++)
        con[i] = salt[i];

      // append dice
      con[l] = (byte)dice;

      var hash = sha.ComputeHash(salt);

      return hash;
    }
```

After bob recives a salt and dice two verify, the previously recieved commitmentHash he uses `MatchesCommitment` which compares the commitmentHash to a new hash generated from a salt and a dice roll

```
  public static bool MatchesCommitment(byte[] commitmentHash, byte[] salt,
  int clientDice)
  {
    return Encode(commitmentHash) ==
  Encode(GenerateCommitmentHash(clientDice, salt));
  }
```

When alice and bob has recived eachothers diceroll they each compute the final dice roll with `ComputeDiceRoll` which combines two dice rolls from each party into one diceroll.

```
public static int ComputeDiceRoll(int clientDice, int serverDice)
{
  return 1 + (((clientDice-1) + (serverDice-1)) % 6);
}
```

Node/TLSSockets

The javascript/node part of my codebase is responsible for creating the secure channel. Where Alice and Bob act as a client and a server, respectively, on the network. The library node:tls sets up a server/client session with two-way authentication.

To enable two-way authentication I pass `requestCert: true` and `rejectUnathorized: true` as options to the `tls.createServer()` function, to require client authentication, which is quite uncommon for normal websites.

```
// server.js
const tls = require('tls');
const fs = require('fs');

const options = {
    key: fs.readFileSync('server-key.pem'),
    cert: fs.readFileSync('server-crt.pem'),
    ca: fs.readFileSync('ca-crt.pem'),
    // Servers may set requestCert to true to request a client certificate
    requestCert: true,
    // The server will reject any connection which is not authorized
    // with the list of supplied CAs
    rejectUnauthorized: true
};
// [...]
```

I use openssl to generate certificates for TLS, which I use as the "they have access to a Public Key Infrastructure" from the assignment description.

Infrastructure

It was a little tricky to interface between the two languages, but now works by forwarding std in/out between the node and c# application. Each language was great at doing each part, and dockerizing everything removed all the platform-specific issues I ran into.

I package my source code for the node and dotnet project, along with keys into a server and a client docker image, from a base image `mcr.microsoft.com/dotnet/aspnet:6.0` with node installed on top of it. The containers are networked together using docker-compose to run on localhost by specifying `network_mode: host`.