Security 1: Mandatory Hand-in 2

Adrian Borup (adbo@itu.dk)

Oct 25 2022

Protocol overview

From the assignment text, it is given that Alice and Bob

1. Do not trust each other to roll randomly if they know the other person's outcome
2. Do not want anyone else to be able to learn anything about the dice game, with communication happening on an insecure network

To solve the first part, we will assume that we have a secure communication channel, which we will get to later.

Rolling a die

The problem with just Alice and Bob simply rolling a die each and sending the result to each other is as follows: if Alice rolls a die, Bob may look at Alice's die and change what he rolls according to her roll. Therefore we need a way for Alice and Bob to roll a die each without them being able to use the other person's roll to influence their own roll. For this, we will use the Coin Tossing algorithm.

Coin Tossing works as follows:

1. Alice initiates the communication by performing these steps:
    1. She rolls a die, with the outcome being denoted a.
    2. Before she shares the outcome with Bob, she generates a random number r. Why we need this will be explained afterwards.
    3. She computes a *commitment* - a way for her to bind herself to the outcome of the roll without sharing the actual roll.
    4. The commitment c is computed as the hash of the concatenation of r and a. That is $c = H(r|a)$.
    5. Alice sends the commitment to Bob
2. Bob receives a commitment c from Alice and performs the following steps:
    1. He rolls a die, with the outcome being denoted b. Because Bob doesn't know a, he can't use it to influence b.
    2. He sends b to Alice.
3. Alice receives Bob's roll b:
    1. She sends the values she used to make her commitment (r, a) to Bob
4. Bob receives (r, a) from Alice:
    1. He computes the commitment himself as $H(r|a)$ and checks if indeed $c = H(r|a)$. If it is, Alice can't have lied about rolling a.
5. Both parties combine their respective rolls into a "final" outcome as such: $(a \oplus b) \bmod 6 + 1$ (with $\oplus$ denoting a bitwise XOR).

This is secure under the assumption that finding another input to the hash function that produces the same hash is infeasible. However, this assumption relies on the fact that the input space is large enough. This is why we need a (sufficiently large) r for the commitment. For a die roll, there can be a grand total of 6 different outcomes: $1 \dots 6$. If we were to calculate the commitment as the hash of the roll alone, it would take at most 6 attempts to reverse the commitment. But if we use a large, random r to encode the commitment, it becomes infeasible to reverse it.

Secure communication

In order for Alice and Bob to perform the protocol, they need a safe means of communication. To fulfil their paranoid requirements, we need to obtain the following three properties:

1. Confidentiality: Nobody else can read and understand the messages sent
2. Authenticity: We must be able to know that a message came from a specific party
3. Integrity: We must be able to determine if a message has been modified since it was sent

In short, I will be using the following to obtain the above properties:

1. Confidentiality
   - AES (symmetric encryption)
2. Authenticity
   - Authenticated Diffie-Hellman key exchange
   - ElGamal digital signatures
   - Asymmetric private/public key pairs
3. Integrity
   - Message authentication codes (MAC)

Why these make the communication secure will be explained soon.

Issues with the protocol

One issue with the protocol is that it results in a biased die roll, which is a rather large issue because the game is all about rolling random dice. The issue occurs when combining the two rolls into a final roll, where an XOR is used. If we send 3 bits over the network (which is the least we need to represent 6 numbers), we can represent 8 different numbers: 0 through 7. But only 1-6 are valid rolls. If two valid rolls are sent over the network, the XOR of them can become a number that is outside the range of 1-6. With our protocol, given the XOR result on the left, the final outcome is the number on the right:

- $0 \bmod 6 + 1 = 1$
- $1 \bmod 6 + 1 = 2$
- $2 \bmod 6 + 1 = 3$
- $3 \bmod 6 + 1 = 4$
- $4 \bmod 6 + 1 = 5$
- $5 \bmod 6 + 1 = 6$
- $6 \bmod 6 + 1 = 1$
- $7 \bmod 6 + 1 = 2$

As shown, the inputs 6 and 7 result in a roll of 1 and 2, which makes 1 and 2 double as likely to be the outcome compared to 3-6. However, as has been mentioned by the lecturer, this is not really important for the sake of this project. It's more about the trust and security in communication.

Why is the protocol secure?

In this section I will briefly outline how the methods I have used make the communication secure. One important note: I assume that we have access to a Public Key Infrastructure (PKI) where we can check that public keys have been signed by some trusted Certificate Authority (CA).

Authenticated key exchange

Keywords: *Diffie-Hellman, ElGamal signatures, public/private keys*

The first step in our communication is to contact each other and establish shared secrets for secure communication. For this, we use authenticated ephemeral Diffie-Hellman key exchange.

The *Authenticated* part means that we know who the sender is and that we can verify that fact. To achieve authenticity, we use ElGamal digital signatures to sign key exchange messages. Messages between Alice and Bob look as such:

$$A \to B : \{A, B, Y_a\}_{sk_a}$$
$$B \to A : \{B, A, Y_b\}_{sk_b}$$

Where Y_a is Alice's "public" part of the Diffie-Hellman key exchange and sk_a is Alice's digital signature for that message. When one of the parties receives a message, they find the public key for the designated sender and use it to verify the digital signature (which can only have been produced correctly by that sender). This way we ensure that:

1. The sender is who they say they are
2. The message is intended for us
3. The public Diffie-Hellman component is generated by the sender and not some man in the middle
4. All of the above are verifiable by the fact that the whole message is signed with the sender's private key.

Otherwise, the Diffie-Hellman key exchange protocol happens as usual, generating the shared secret. It's worth noting that ephemeral Diffie-Hellman should be used, with ephemeral meaning that the "private" keys are generated anew for each exchange rather than being static secrets. This ensures *forward secrecy*: if a secret is leaked, it only compromises one session, not all sessions ever created.

Confidentiality

Keywords: *AES*

Using our authenticated shared secret, we can encrypt and decrypt messages using a symmetric encryption algorithm. Here we just use the industry standard AES (Advanced Encryption Standard), which has been proven secure.

Integrity

Keywords: *MAC*

We also want to be able to ensure that our encrypted messages haven't been modified in transit. We can achieve this by using a Message Authentication Code (MAC). Once we have our message ready, we calculate a hash of it before sending it. We append this hash to the message, encrypt it, and send it. If a single bit is changed in the message, the hash will be different. The recipient then extracts the MAC, calculates the hash of the remaining message, and compares it to the MAC. If they match, we know that the message has not been modified.

Final remarks

The security of the above methods depends on the size of the primes and random bits used. Of course one should aim to use sufficiently large primes and random bits that have been proven secure by today's standards.

The methods used in the protocol have been inspired heavily by TLS. That being said, it would certainly be smarter (and likely far more secure) to use TLS directly instead of implementing these things myself. I only did it to learn how it works.

Implementing the protocol

The implementation has been written using Python (version 3.9.13) using communication via TCP/IP, and the code lives within the `src` directory handed in alongside this report.

What follows is an overview of the files:

- `player.py`: the entry point and the main logic the game itself
- `pki.py`: a wrapper that acts as our PKI
- `key_exchange.py`: performing an authenticated key exchange
- `signature.py`: creating and verifying signatures
- `encryption.py`: a wrapper for encrypted and MAC-validated network communication

- `network.py`: a wrapper for communicating over the network
- `constants.py`: the primes and random bit sizes used

For instructions on running the program and seeing sample output, see the README file. To view it online, you can find the repository at https://github.com/adbo-ITU/ITU-SECU2022-MH2.