Security mandatory hand in 2

Villum Sonne

Designing a protocol:

Design a protocol that allows Alice and Bob to throw a virtual 6-sided dice over the insecure network even though they do not trust each other without allowing for an adversary to see that they are playing dice.

The protocol goes as follows:

1. Alice and Bob both agree on a cipher suite.
2. Alice and Bob already know each other's public keys.
3. Alice rolls a dice, and sends the number of eyes *a* and sends a hash-based commitment Com(a,r), which is signed and then encrypted, to Bob:

   $A \rightarrow B: \{ Com(a,r) \} SK_{Alice}, PK_{Alice}$

4. Bob rolls a dice and sends the number of eyes *b* which is signed and then encrypted, and then sends it to Alice:

   $B \rightarrow A \{b\} SK_{Bob} PK_{Bob}$

5. Alice sends (a,r), which is signed and then encrypted.

   $A \rightarrow B: \{ (a,r) \} SK_{Alice}, PK_{Alice}$

6. Both Alice and Bob now know each other's rolls, and Bob is assured that Alice has not changed her roll after seeing Bob's, by comparing Alice's first message with the Com(a,r) of what she just sent.

7. Both Alice and Bob compute the output as (a XOR b % 6) + 1.

This ensures that Alice and Bob can decide on a dice roll based on both of their dice rolls. Choosing the winner is not something the protocol does, but up to Alice and Bob to decide beforehand, e.g., that roll 1-3 means Alice wins while roll 4-6 means that Bob wins the house, children, and cat.

Villum Sonne
Security hand in 2 25-10-2022

Validity of the protocol:

Explain why your protocol is secure using concepts from the lectures.

It is obviously important that for the protocol to uphold security principles, then Alice and Bob must agree on a suitable cipher suite. Their encryption schema must not be weak, and their hashing algorithm must be valid as this ensures that it is *hard* to find the pre-image of the hash. Furthermore, it is also assumed that Alice and Bob are already aware of each other's public keys, as this would otherwise just add steps to the protocol, which doesn't serve its actual purpose. The nature of the problem means that it has been obvious to draw inspiration from Blum's coin tossing protocol[1] to design my own protocol. Blum's protocol is essentially a more abstract version of my own protocol, which by definition means that all properties of Blum's protocol is also present in my protocol.

The main argument for Blum's algorithm holding is that it is impossible for either Alice or Bob to know what the other one has rolled (or said they rolled) before sending their own roll to the other person. This also holds for my protocol on the basis of Alice's initial commitment, which makes it possible for Bob to check if Alice's has changed her roll after seeing what Bob has rolled. This commitment ensures the *integrity* of Alice's initial roll, since she is not able to alter the data to her own advantage.

The use of encryption and signatures also makes sure the protocol is secure in a Dolev-Yao environment. Encryption ensures *confidentiality*, that only those authorized can read messages, while signatures ensures *integrity*, that the messages have not been altered by someone else than the sender. However, availability is not guaranteed, as it is possible for an adversary to perform either denial of service attacks or withhold messages between Alice and Bob.

The algorithm for determining the roll goes as follows:

$$(\text{a XOR b } \% \ 6) \ + \ 1$$

XOR ensures that it is not so trivial to try to game the outcome to one's own advantage. The modulo 6 operation is necessary since it requires 3 bits to represent 6 (110). This means that a XOR operation: 110 ^001 can generate the output (111) which is 7, which is not a valid dice roll. Furthermore, the modulus operator returns a number between 0 and 5, and therefore the result is incremented.

[1] https://dl.acm.org/doi/10.1145/1008908.1008911

Villum Sonne
Security hand in 2 25-10-2022

Implementation of protocol:

The implementation of the protocol can be found in the folder ‘src’. It contains two files: client.py and main.py. To test the protocol you simply run the main.py file by writing *python3 src/main.py* in the root folder. Note: It is necessary to have the python packages *twisted* and *pycryptodome* installed. All information can be found in the readme.

The *twisted* package is used to make the two clients talk to eachother over the local network. The details are not interesting for this course.

The *pycryptodome* package provides an implementation of El Gamal encryption and decryption, as well as signatures. The package also provides the SHA256 hashing algorithm, which is secure enough for the purpose of this assignment. The SHA256 algorithm is however prone to collision attacks.

Whenever a client sends a message, this message is first signed with this client’s private key, and then encrypted with the public key of the other client. I found it necessary to send the encrypted message and the signature in two separate messages, separated by half a second. I couldn’t figure out how to send structs/classes with the twisted package and opted for this solution, even though it is less secure. The verification of a signature however still relies on the last message received, so it is not possible for an adversary to just send a fake signature.