Mandatory Handin-2
Security I

Joakim Hey Hinnerskov
IT university of Copenhagen

December 29, 2020

Question 1 *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 othe*

My design of the protocol assumes that Alice and Bob knows each others Public-keys.

Dice roll protocol:
- A generates random bit a and random 128 bit string r.
- A concatenates a and r and hashes the concatenation computing c.
- B generates random bit b.

1. $A \rightarrow B : \{c, sign\}PK_b$
2. $B \rightarrow A : \{b, sign\}PK_a$
3. $A \rightarrow B : \{(a,r), sign\}PK_b$

- B hashes (a,r) verifying that no changes has been made to the message
- A & B locally computes $a \oplus b$, converts it to integer representation and takes it under $mod\ 6+1$.

Question 2 *Explain why your protocol is secure using concepts from the lectures.*

This protocol is expecting that atleast one of the involved parties is honest. Thus a or b is expected to be a random bit.

Step 1. This message is encrypted with B's Pubic-key, thus it is only possible for B to decrypt the message and therefore ensuring confidentiality. Furthermore A signs the message and B is able to verify that the message originated from A and has not been tampered with, therefore ensuring authenticity. At last, hashing of the concatenation (a,r) resulting in c, ensures that B cannot learn anything about a before sending b. Concatenating a with a random 128 bit string ensures that B is not able to bruteforce different $a's$ such that $a = H(c)$.

Step 2. Only A can decrypt the message and authenticity is ensured with B's signature. It is irrelevant that A might learn something about b as A cannot change the original message.

Step 3. A sends (a,r) such that B can perform the hashing to verify that a was not changed. This is possible because hashing always computes the same result. Thus when the commitment of the concatenated bit a and bit string r is hashed, the new hash that B computes, has to be the same as the original. Each party is now able to locally compute $a \oplus b$ resulting in a random bit string, then converting it to the integer representation of the bit string and taking this under *mod 6 + 1* will imitate a dice throw.

Important factors This protocol works because *XOR* on two bit strings will return a random bit string as long as either a or b is random, thus one party has to be honest. Authenticity is an important factor and the signatures ensures that A or B knows that the message is from the correct sender. Furthermore the same encoding and hash algorithm ensures that we are able to verify messages. Randomness ensures that one party cannot learn something about the message before sending their random bit.

Question 3 *Implement your virtual dice protocol in a programming language of your choosing. The implementation must consist of a program representing Alice and another program representing Bob that communicate over a network (two processes running on localhost is ok). You can use any libraries or programming languages of your choosing.*

To implement the virtual dice protocol I used the *Socket* library, setting Alice up as the client and Bob up as the server. For the cryptography I used the *Cryptography* [1] library, but the hazardous module, which I found suitable for this exercise, because the assignment stated that Alice and Bob knows each others Public-keys. Both Alice and Bob imports my own *MyCryptoLibrary* just to make the code more readable, as they are using the same functions several times.

Program structure:

```
src
├── alice_local.py
├── bob_local.py
└── lib
    └── MyCrytpoLibrary.py
```

Alice and Bob starts out by generating a key-pair with RSA and, in this program, saving their public key in PEM format to the folder (imitating that they already got each others Public-Keys). Then they both open each others Public-keys and serializes the PEM formattet key into the program and saves it such that the library can use the keys. Then Alice connects to Bob and initiates the protocol.

RSA is used for the encryption to ensure confidentiality and also for signatures to ensure authenticity. SHA256 is used for hashing with utf-8 encoding to ensure that they compute the same hash.

Execution:

To execute the program run *bob_local.py* and then *alice_local.py* with python3, the program and protocol will then execute sequentially. For test purposes Alice's a is **not** random, but as Bob's b is, the dice throw computed will be random. Changing both a and b to be hard coded results in the same dice throw each run.

[1] The documentation helped me to use this library `https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/`