Rolling dice

Fabio Massimo Ercoli *

December 30th 2024

1 Protocol basic ideas

Most of the time, in security scenarios Alice and Bob collaborate implementing security strategies against possible attackers (such as Eve). This case is quite different since Alice and Bob are competing each other, and they can try to trick each other.

Non repudiation In my opinion a crucial security property to implement the protocol is the *non repudiation*. We want to be sure that once a dice is cast by a given player (e.g., Alice and Bob) the outcome must not be disputable. In particular, we want that all the entities knowing the public key of the caster are able to check if a given action was really made by the caster, with no possibility of making mistakes, with the only exception, not considered in this protocol, that the private key of the caster has been compromised.

The (non)randomness of the casts Cast outcomes should be produced in such a way their values are taken uniformly at random in the 1 to 6 interval. There are probably several ideas that can be used to try to enforce it. My idea here is to produce two numbers for each cast of Alice and Bob, one from Alice and one from Bob, and then xor-ing them.

- If both Alice and Bob produce both random numbers, of course the xor of two random numbers is a random number, so it is fine.
- If one of them produces a random number and the other tries to trick producing some number that is not random, and the one that produces the non-random value does not know the value produced by the other before casting the dice, again we're in a good shape, since the result will be a random value again.
- If both try to trick producing some non-random values, if none of them know the non-random value produced by the other before to cast her/his

*More on me: my GitHub page - my Linkedin page.

choice, even if the result is not random, I think that it is impossible to get some advantages for the trickier to have more chances to win.

Synchronization According to what we said in the last paragraph we need to synchronize the casts so that none of the players can see the cast of the other player before to cast her/his dice. The solution we adopt here is to use a third party process to collect both the contribution related to a single pair of casts, and only when it receives the the contributions for both the players, it can share them with the same players.

About the security for this third actor, we can call it Carol, we assume that it is safe, maybe it is a bastion host. In any case I think that the protocol can be extended adding for instance the certificate of this third party server, but I don't do that here for sake of simplicity.

2 Proposed protocol

2.1 Assumptions, hipothesys, limitations

First of all we assume that:

- The Alice node, the Bob node, the Carol node correctly have the public key of Alice and Bob. Imaging that they have safely obtained them using PKI.
- The Carol node is trusted.
- We don't expect that the rolls of the dices is uniformly at random in all cases. We expect it in most of the cases and then this does not happen, we expect that the player that produced the non-random value didn't get any advantage from it.

2.2 Protocol Data structure

We need to imagine a data structure of an Object Oriented class to denote (and store all the state of) a *Game*.
The *Game* entity contains:

- **id**: An identifier of the game, maybe an UUID. This value is used to identify a game instance unequivocally.
- **aliceRolls**: A list (ordered) of what we can call *RollDice* items for Alice
- **bobRolls**: A list (ordered) of what we can call *RollDice* items for Bob

Where the size of the lists is equal to the size of k (the number of dice rolled).
The *RollDice* entity contains:

- **roll**: A random generated value that denotes the player roll.
- **antiRoll**: A random generated value that will be xor-ed with the other player's roll in the same position of the *RollDice* item lists.

2.3 Compute the rolling dice outcomes and the winner

The roll dice outcomes and then the winner can be computed only if we have both the roll dice items from Alice and Bob.

The length of the two lists *aliceRolls* and *bobRolls* must be the same (k). For each i position from the first to the last we compute the values xor-ing the roll values with the corresponding *antiRoll* of the other player, then we compute the module 6 and we add 1 to get an outcome in the interval [1 to 6]:

- the i-th Alice roll outcome = (*aliceRolls*[i].roll $\oplus$ *bobRolls*[i].antiRoll % 6) + 1
- the i-th Bob roll outcome = (*bobRolls*[i].roll $\oplus$ *aliceRolls*[i].antiRoll % 6) + 1

The computation of the winner is trivially done summing the roll outcomes of both players and selecting the player who has the higher sum.

2.4 Protocol dynamic

We can describe the dynamic of the protocol with the following steps:

- When the game starts, Alice and Bob need to agree on the same game id.
- Alice computes the *aliceRolls*,

  she signs using her private key the *aliceRolls* + the game id

  she sends the list tougher with the game id and the sign to Carol
- Concurrently, Bob computes the *bobRolls*,

  he signs using his private key the *bobRolls* + the game id

  he sends the list tougher with the game id and the sign to Carol
- When Carol delivers a pair of *aliceRolls bobRolls* lists, for which she has verified the sign:

  She sends *bobRolls* tougher with the game id and the Bob's signature to Alice

  Concurrently, she sends *aliceRolls* tougher with the game id and the Alice's signature to Bob
- When Alice delivers the *bobRolls* is able to verify the authenticity and compute the rolling dice outcomes and the winner and the same symmetrically is true for Bob.

At the end of the process neither Alice nor Bob can repudiate their rolls, this guarantees the consensus on the rolling dice outcomes and the winner for the given game.

3 Extra: Third party role

In this implementation of the protocol the role of Carol is very limited, she basically only implement an exchange pattern for the messages containing the lists provided by Alice and Bob.

We could have provided more responsibilities to this component, for instance we could have generated the random numbers from Carol node. I didn't do that because usually limiting the load on third party nodes is a good practice to improve the scalability of the solution, since in this way we spread more the computation costs among different nodes.