CMPE483-Autonomous-Decentralized-Lottery

Implementing an autonomous decentralized lottery with a Solidity smart contract.

Project Details

That's a project I've created at university homework for the course, CMPE 483 - Blockchain Programming on Dec 6th, 2019

What the project is about can be found via Project1.pdf (./Project1.pdf)

In summary, the objective was to create a decentralized lottery with a Solidity smart contract such that,

- It uses a ERC20 (EIP20) Contract (https://github.com/ConsenSys/Tokens/blob/master/contracts/eip20/EIP20.sol) to use TL tokens
- Every 2 week a new lottery is started
    - In the first week, any user can purchase a ticket by exchanging 10 TL tokens.
        - When purchasing a ticket each user sends a random number.
        - The sent random number should not be stored on blockchain, the user should keep it secret.
    - In the second week, users who purchased tickets should reveal their numbers.
        - Users who do not reveal their secret numbers are disqualified (the tickets with not revealed secret numbers are lost)
- After a lottery is finished it awards the collected TL tokens on that lottery to the participants.
    - If `M` is the total collected TL tokens and there are `N` valid tickets (tickets with revealed secret numbers) there would be `ceil(log2(N))` prizes.
    - Each prize is calculated as follows: (Making sure all collected tokens are awarded)
    - `i=1,2,...ceil(log2(N))` `Pi = floor(M/2^i) + (floor(M/2^(i-1)) MOD 2)`
    - A ticket may win more than one prizes.
    - Each prize should be awarded to one ticket picket fairly randomly.
        - See Discussion about generating random numbers (https://ethereum.stackexchange.com/questions/191/how-can-i-securely-generate-a-random-number-in-my-smart-contract)
    - Each ticket can be redeemed anytime after a lottery finishes.

My Solution

I've created a Solidity contract which contains three callable functions, `purchaseTicket`, `revealNumber` and `redeemTicket`

The most challenging part was to rewarding prizes randomly and fairly. My solution to this was:

- Each submitted `randomNumber` must be a `32 bit unsigned integer`
- Each ticket has a hashed random number submitted by user.
    - It hashes it with `msg.sender`, `lotteryNumber`, `ticketNumber` concatenated to the `randomNumber`
    - `msg.sender` is to make sure only the owner can reveal it.
    - `lotteryNumber` is to prevent malicious users to guess the submitted number through comparing previously submitted tickets of other persons.
    - `ticketNumber` is to prevent malicious users to check whether a user submitted the same random number in one lottery twice.
- In this stage it is not possible to estimate how many tickets will be submitted, and what the random number would become when all submitted random numbers are combined.
- I've put a hard limit of `2^31 - 2` tickets at most to be submitted for reasons I'll explain soon. (It's possible to increase this limit, but since no requirement was specified about it, I've chosen that limit)
- In the second stage, users reveal their secret numbers, and the contract XORs them to obtain a `randomNumber` for the lottery.
- In this stage it is not possible to estimate how many tickets will be revealed in the end, but it is possible to find an upper threshold as the tickets purchased in previous stage.

After the second stage (revealing stage) is finished, no one can make any change in that lottery round's randomness and the created XOR of all submitted random numbers are fairly random in range `[0, 2^32 - 1]`

In the redeem stage I had to award more than one prizes using only one initial random number:

- I chose to use a PRNG [Park-Miller-Carta Pseudo-Random Number Generator (http://www.firstpr.com.au/dsp/rand31/#History-PMMS)](http://www.firstpr.com.au/dsp/rand31/#History-PMMS) giving the initial seed value as the initial random number I've obtained from XOR
- `(seed * 16807 mod(2^31 - 1))` is the main logic of that generator.
- `16807` is a `full-period-multiplier` in mod `2^31 - 1`, meaning that each number in range `[1, 2^32-2]` appears exactly once in a sequence of generated numbers starting with any seed in that range.
- Thus, the seed should be always in range `[1,2^32-2]`, which made the initial seed picking very hard.
- That's because, in the beginning I get a fairly random number in range `[0, 2^32 - 1]` but choosing a fairly random seed in range `[1, 2^31-2]` is impossible.
- The simplest seed picking would be to `seed = (randomNumber % 2^31-1)+1` which would give an unfair advantage as `(2^32) % (2^31-1) = 2`, the chances of `seed = 1` and `seed = 2` being picked is one more times than each other seed.
- More precisely each seed would be able to be picked `floor((2^32) / (2^31-1)) = 2` times while `seed = 1 | 2` can be picked `ceil((2^32) / (2^31-1)) = 3` times.

To overcome that problem I've added a special condition if the `randomNumber == 2^31-1` or `randomNumber == 2^31-2` which are the extra cases causing an advantage to seed `1` and `2`

- If the randomNumber is one of those, instead of using it to determine seed, I use the number of valid tickets in the contract, which is also pretty random.
- Let's say that I award each prize to ticket with `winnerTicket = (seed - 1) % validTicketCount` until I say otherwise.
- Yet, if I picked the seed as the `validTicketCount` directly, that'd give an unfair advantage to the last revealed ticket such that the first prize would always go to the first ticket.
- To prevent it I've used `2^31-1 - validTicketCount` which maps to range `[1, 2^31 - 2]` (ignoring the case when no valid tickets are there, since redeem function would become uncallable in that case)
- Which gives the advantage to first users if there're extremely many valid tickets but gives that advantage to a random ticket since it's very hard to estimate the first winner since `((2^31-1-validTicketCount) - 1) % validTicketCount` as its result changes the winner randomly as well.
- In small validTicketCount such scheme is fairly random:
    - if validTicketCount is 50 the initial seed is 2147483597 and the first winner is 46
    - if validTicketCount is 49 the initial seed is 2147483598 and the first winner is 42
    - if validTicketCount is 51 the initial seed is 2147483596 and the first winner is 25
- In bigger validTicketCount however it is not and it favors the first revealers:
    - if validTicketCount is 2147483590, the initial seed is 57 and the first winner is 56
    - if validTicketCount is 2147483591, the initial seed is 56 and the first winner is 55
    - if validTicketCount is 2147483589, the initial seed is 58 and the first winner is 57
- But, with a tiny modification it is possible to make it all fairly random again:
- Instead of using `winnerTicket = (seed - 1) % validTicketCount` directly, I use `seed = (seed * 16807 mod(2^31 - 1))` once in the beginning.
    - if validTicketCount is 50 the initial seed is 2147483597 and the first winner is 46
    - if validTicketCount is 49 the initial seed is 2147483598 and the first winner is 42
    - if validTicketCount is 51 the initial seed is 2147483596 and the first winner is 24
    - if validTicketCount is 2147483590, the initial seed is 57 and the first winner is 957998
    - if validTicketCount is 2147483591, the initial seed is 56 and the first winner is 941191
    - if validTicketCount is 2147483589, the initial seed is 58 and the first winner is 974805
- Which makes it impossible to estimate the first winner in a localized manner such as (the first ones revealed) but instead changes the outcome randomly with every extra ticket revealed.

- Now, there's only this tiny little problem left: Since the case when only 1 valid ticket exists is meaningless, the (2^31-1-validTicketCount) maps to [1,2^31-3] while our initial seed expects [1,2^31-2].
    - An unfair disadvantage to one/two seed numbers are not a big problem when compared to an unfair advantage to one/two seed numbers.
    - So, I decided to share this disadvantage of seed 2^31-2 with 2^31-3 by adding an if statement as such:

      ```
      if (seed == 2**31-3) {
          seed += currentRandom - 2**32-2;
      }
      ```

    - I took advantage of the initial random number's 2 extra cases to switch between 2^32-2' and2^32-3`
- Overall that makes the seed chances as below:
    - seeds: [1,2^31-4] => 9.313225759e-10 = .0000000009313225759
    - seeds: [2^31-3, 2^31-2] => 9.313225757e-10 = .0000000009313225757
- For more details and calculations please check out the source code: src/contracts/LotteryContract.sol:188-221 (./src/contracts/LotteryContract.sol)

Testing

I've used `truffle` in combination with *ganache-cli* to make automated tests. `truffle` is a javascript library allowing automated testing with MochaJS for Solidity contracts.

My test scripts are stored in src/test/ (./src/test);

- src/test/eip20.js (./src/test/eip20.js) - I tested the initial TL Token balance of the first account
- src/test/lottery.js (./src/test/lottery.js) - I tested errors in different scenarios and a happy path in which 5 people attend to a lottery, one forgets to reveal his number and the other 4 redeems all prizes.
- src/test/random-lottery.js (./src/test/random-lottery.js) - I tested 5 rounds of lottery with up to 100 randomly chosen tickets in each round, all rounds are completed successfully and all participants redeemed their prizes to their eip20 contract account.

For more details, please check the test scripts.

To automatically run those use `test.sh` in the top level directory of this project.

```
chmod +x ./test.sh
./test.sh
```

That script requires you to install `truffle`, *ganache-cli* and `concurrently` which you can install allwith running `npm i -g truffle ganache-cli concurrently`