Optimize RPS

Problem

1. Front-running
2. Currently use idx to identify the player (player needs to send idx along with choice)
3. Tokens can be stuck on the contract in cases
   1. If there is only one player. They'll have to wait forever for another player to join.
   2. If only 1 player reveals their choice.
4. Can play 1 time after being deployed.

Solution

Front running

With front running, we can commit and reveal to ensure that other players won't know the player's choice before revealing.

1. In this project, I decided to edit CommitReveal.sol to make the function internal and change the datatype for simplicity. commit
2. Change the input function to accept a hash of choice(bytes32)
3. Make sure that player can reveal their answer after all players commit their choice.
4. Then after everyone reveals their choice, Check for the winner.

```
function input(bytes32 hashChoice) public {    // input accept hash choice
  require(numPlayer == 2);
  require(numInput < 2);
  ...
  commit(hashChoice);                          //commit has choice
  ...
}

function revealChoice(uint choice, string memory password) public {
  require(numInput == 2);                      // can reveal after everyone
already commit choice
    ...
    if (numReveal == 2) {                        // if everyone reveal answer ->
check for winner
      _checkWinnerAndPay();
    }
  }
```

Identify the player

I use the sender's address as an identifier of the player. It's hard to scale a number of players up, but that's not the case here.

Tokens can be stuck on the contract

1. We can have a function in case there is only one plyer nad they want to retrieve a token and end the game round.
2. We can do timed commitment so if players are unwilling to commit or reveal a choice in time, other players can claim all tokens. (In this case, there are 2 players so others will get all the money)

```
function claimReward() public {
  require(msg.sender == player0.addr || msg.sender == player1.addr);
  address payable account = payable(msg.sender);
  Player memory p;
  if (msg.sender == player0.addr) {
    p = player0;
  } else if (msg.sender == player1.addr) {
    p = player1;
  }
  // if there are no other player, the player can claim the reward
  if (numPlayer < 2) {
    account.transfer(reward);
  }
  // if the others player has not input the choice, the player can claim the
reward
  else if (numInput < 2) {
    require(block.timestamp > inputDeadline);
    require(p.choice == unrevealChoice && p.commit != false);
    account.transfer(reward);
  }
  // if the others player has not reveal the choice, the player can claim the
reward
  else if (numReveal < 2) {
    require(block.timestamp > revealDeadline);
    require(p.choice != unrevealChoice);
    account.transfer(reward);
  }
  reward = 0;
  _resetStage();
}
```

Can play 1 time after being deployed.

Since there is no resetting, so we can't replay this contract. We can have the resetStage function to reset the value and call it after finding a winner or someone claiming the tokens.

```
  function _resetStage() private {
    player0.addr = address(0x0);
    player1.addr = address(0x0);
    numPlayer = 0;
    numInput = 0;
    numReveal = 0;
    inputDeadline = 0;
    revealDeadline = 0;
  }

  function _checkWinnerAndPay() private {
    ...
    _resetStage();
  }

  function claimReward() public {
    ...
    _resetStage();
  }
```

Additional

Timed commitment

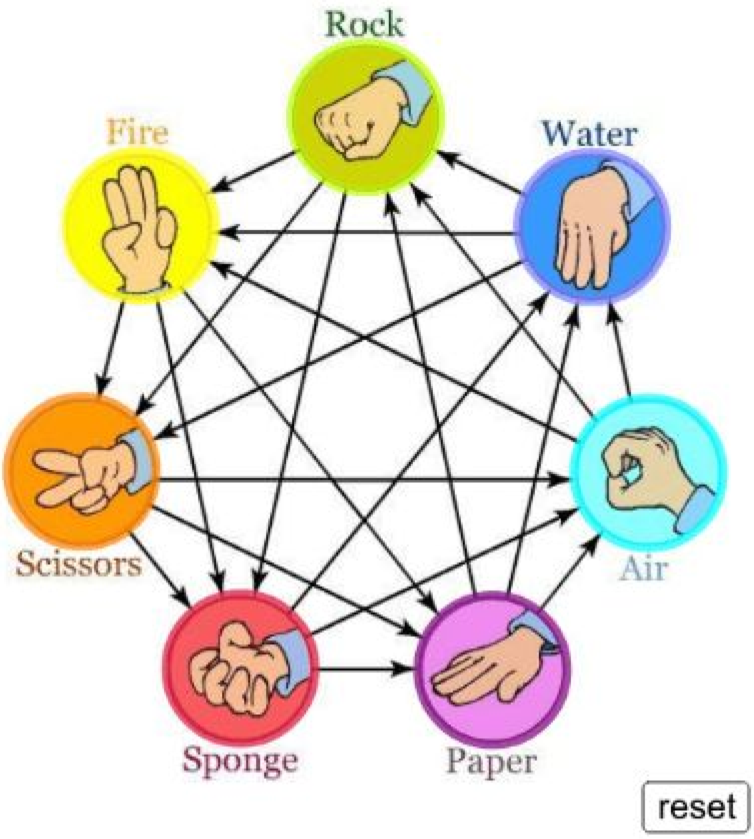
Since we do timed commitments as mentioned. I also provided a timeLeft function for players to know the stage and time left for commit or reveal.

```
function timeLeft() public view returns (string memory stage, uint time) {
  if (numPlayer < 2) {
    return ("Wait for players", 0);
  } else if (numInput < 2) {
    if (inputDeadline > block.timestamp) {
      return ("Time left to input", inputDeadline - block.timestamp);
    }
    return ("Exceed input time", 0);
  } else if (numReveal < 2) {
    if (revealDeadline > block.timestamp) {
      return ("Time left to reveal", revealDeadline - block.timestamp);
    }
    return ("Exceed reveal time", 0);
  } else {
    return ("Game over", 0);
  }
}
```

Extended choice

I also add more choice for player.

```
0 - Rock,
1 - water,
2 - Air,
3 - Paper,
4 - sponge,
5 - Scissors,
6 - Fire,
7 - unrevealed
```



As there are more choices, we need to change the rule for the winner.

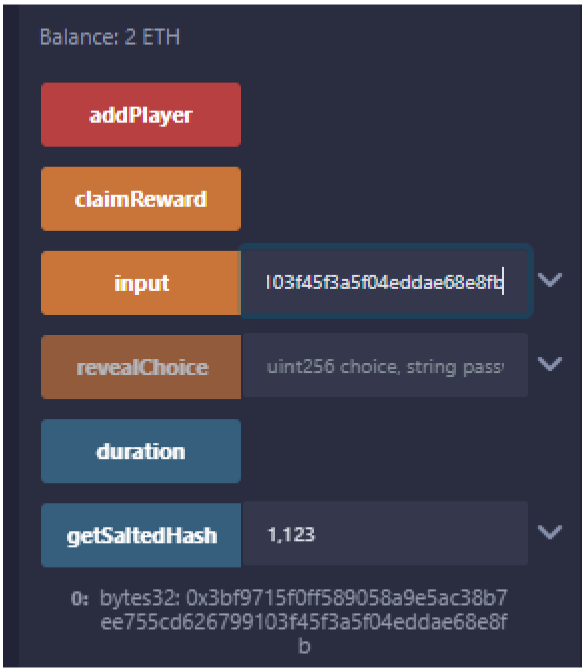
```
function _checkWinnerAndPay() private {
  uint p0Choice = player0.choice;
  uint p1Choice = player1.choice;
  address payable account0 = payable(player0.addr);
```

```
    address payable account1 = payable(player1.addr);
    if (p0Choice == p1Choice) {
      // to split reward
      account0.transfer(reward / 2);
      account1.transfer(reward / 2);
    } else if (
      ((p0Choice + 1) % unrevealChoice) == p1Choice ||
      ((p0Choice + 2) % unrevealChoice) == p1Choice ||
      ((p0Choice + 3) % unrevealChoice) == p1Choice
    ) {
      // to pay player[1]
      account1.transfer(reward);
    } else {
      // to pay player0
      account0.transfer(reward);
    }
    reward = 0;
    _resetStage();
  }
```

Example

example 1 | player 1 chose 1(water), player 2 chose 5(Scissors) => Water rusts Scissors so player 1 should win

1. After adding 2 players, each player will send a hash of choice and salt which can be obtained from the getSaltedHash function as shown.

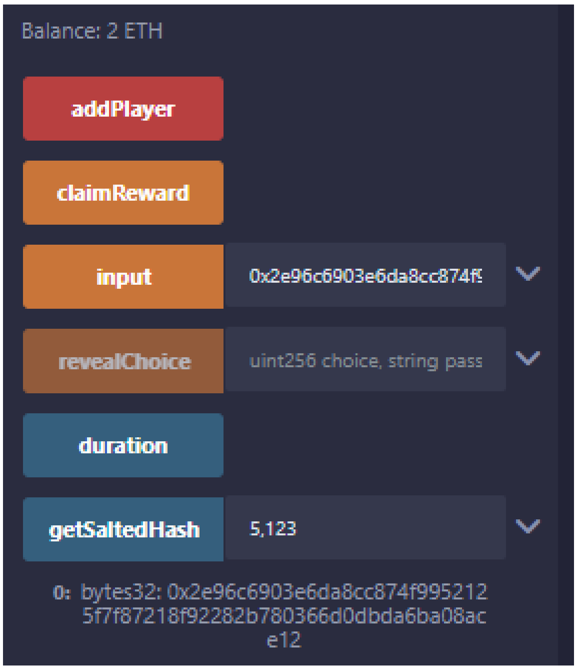


input: l03f45f3a5f04eddae68e8fb

revealChoice: uint256 choice, string pass

getSaltedHash: 1,123

0: bytes32: 0x3bf9715f0ff589058a9e5ac38b7ee755cd626799103f45f3a5f04eddae68e8fb



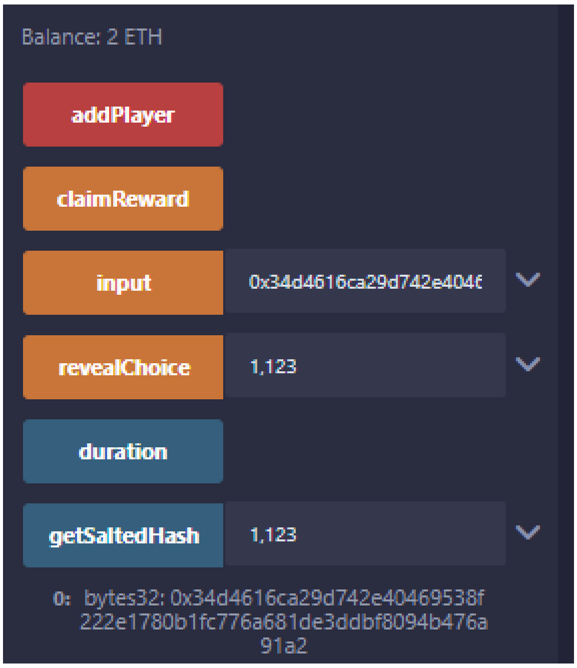
input: 0x2e96c6903e6da8cc874f9

revealChoice: uint256 choice, string pass

getSaltedHash: 5,123

0: bytes32: 0x2e96c6903e6da8cc874f9952125f7f87218f92282b780366d0dbda6ba08ace12

2. Reveal choice



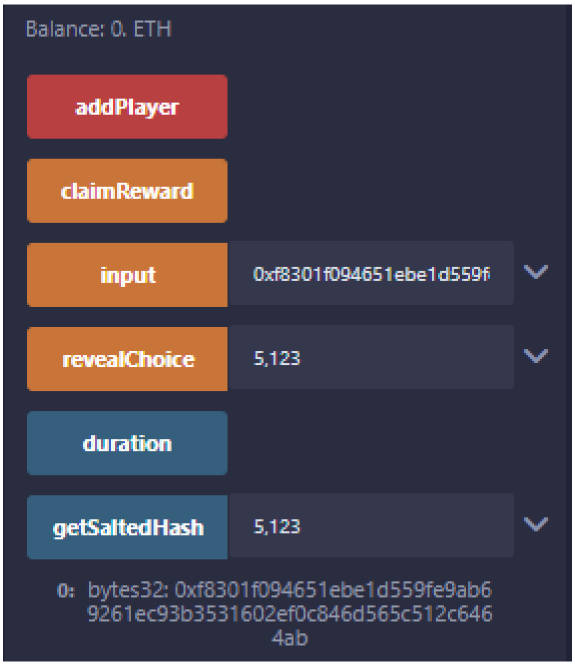
Balance: 2 ETH

input 0x34d4616ca29d742e404

revealChoice 1,123

getSaltedHash 1,123

```
0: bytes32: 0x34d4616ca29d742e40469538f
222e1780b1fc776a681de3ddbf8094b476a
91a2
```



Balance: 0. ETH

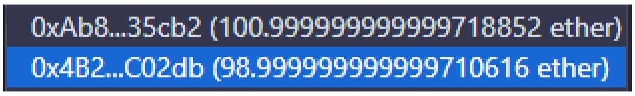
input 0xf8301f094651ebe1d559f

revealChoice 5,123

getSaltedHash 5,123

```
0: bytes32: 0xf8301f094651ebe1d559fe9ab6
9261ec93b3531602ef0c846d565c512c646
4ab
```

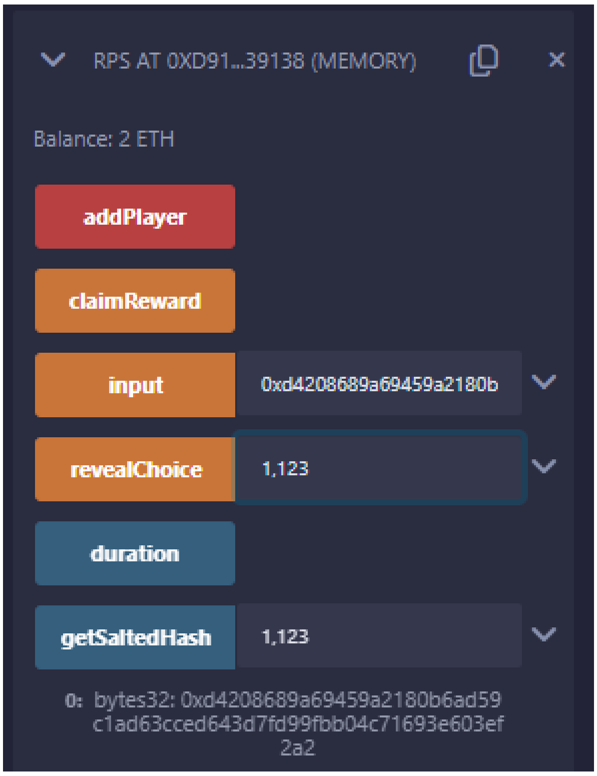
3. Result in player 2 losing 1 ETH to player 1



```
0xAb8...35cb2 (100.999999999999718852 ether)
0x4B2...C02db (98.999999999999710616 ether)
```

example 2 | player 1 and 2 chose 1(water) => It should be even and they get ETH back

1. After adding 2 players, each player will send a hash of choice and salt which can be obtained from the getSaltedHash function as shown.



RPS AT 0XD91...39138 (MEMORY)

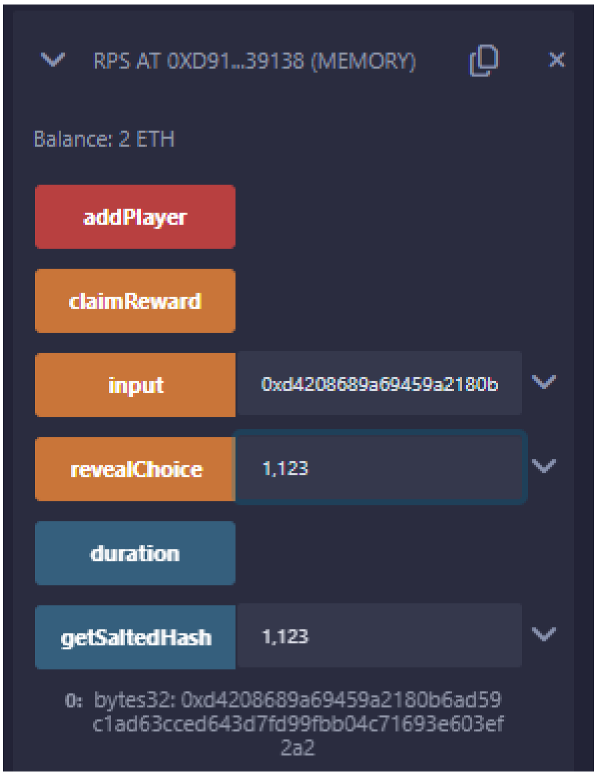
Balance: 2 ETH

input 0xd4208689a69459a2180b

revealChoice 1,123

getSaltedHash 1,123

```
0: bytes32: 0xd4208689a69459a2180b6ad59
c1ad63cced643d7fd99fbb04c71693e603ef
2a2
```



RPS AT 0XD91...39138 (MEMORY)

Balance: 2 ETH

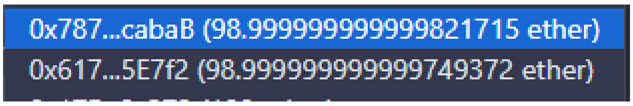
input 0xd4208689a69459a2180b

revealChoice 1,123

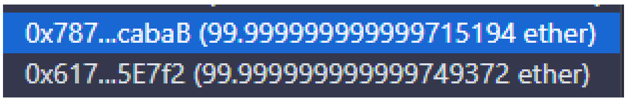
getSaltedHash 1,123

```
0: bytes32: 0xd4208689a69459a2180b6ad59
c1ad63cced643d7fd99fbb04c71693e603ef
2a2
```

2. After revealing answers since the results even each player will get 1 ETH back.



```
0x787...cabaB (98.999999999999821715 ether)
0x617...5E7f2 (98.999999999999749372 ether)
```



```
0x787...cabaB (99.999999999999715194 ether)
0x617...5E7f2 (99.999999999999749372 ether)
```