Assignment 2
S1935167

1: High-level decisions for the design.

My smart contract is used to emulate a dice roll game. Here is a description of the design of my contract.

Rule description.

According to the coursework description, the winner will get reward, and in my contract, the reward is paid by the loser. Since in each game, the loser has a chance to lose 1~3 ETH. So, before each game, the 2 players must have 3 ETH stored at contract to start the game, and after game finish, the winner can take the reward from the contract. Here we use pull over push for reward taking. And we use a struct called player to store the player's information and he's current balance, details about how to do this and how the game process will be explained later.

My contract has these main processes to play a game for new players.:

1.1 Registration.
1.2 Commit seeds.
1.3 Join a game.
1.4 Validate seeds and generate new seeds for dice roll.
1.5 Start game
1.6 Collect reward.
1.7 Add balance to your account(optional).

1.1 Registration

If the player is a new player, the first thing he needs to do is do registration. This process is implemented in a function called register. In this function, the player stores his information and send at least 3ETH to the contract as his initial balance. The reason why my contract needs a registration is my contract uses pull over push for reward taking, by recording each player's account information, we can simply store the reword into the player's account balance, and players can play many games and then withdraw once in the end, they do not need to withdraw their reward after every game, this makes the contract more friendly to use and more efficient. And record each player's information in a player struct can make further data processing more convenient, and make the code looks clearer.

1.2 Commit seeds

After the player successfully registered, he can start to play games. But before that, the first thing he need to do is commit his seeds. This is the way I implemented to achieve fair of the game, more details about how to achieve fair will be explained in "How is it guaranteed that a player cannot cheat?" part later, here I just writes some high-level description about how the process go through. To explain this, I need to explain how to generate dice result in my

contract. When the contract start to roll dice, the information it has is: **1**: 2 seeds from **2** players (these 2 seeds is not the original seeds they sent at commit stage, I will explain this in validation part), 2:block.timestamp, **3**:function's caller's address(msg.sender), and **4**:block number (these 3 data is just used to add some extra randomness, they are not used to prevent cheat). The thing I do is merge all these data and hash the result. Then mod the hash by using %6, this will generate a number with value 0~5, and then I add 1 to the result. It will give me a number from `1~6 to represent the dice result. The commit is achieved by a function: commit_seed, it needs the user send the seed with data type: byte32 as function input. So how does user get his seed? Firstly, the user needs to decide a secret string, which can be any value, and record this secret carefully since it will be used for seed verify later. Then the user needs to preprocess the secret, which is apply abi encode and keccak256 hash on it. This can be achieved by using some online tools, for example, https://adibas03.github.io/online-ethereum-abi-encoder-decoder/#/encode can be used to do abi encode, https://emn178.github.io/online-tools/keccak_256.html can be used to do keccak256 hash(remember to choose input type as Hex instead of text). Then, the player can get the input value which is: 0x + "the value gets from the online tools". He can send the seed he calculated by using commit_seed function, once he done this, his game status will change to committed, which will allow him to do further process.

1.3 Join a game.

This process is implemented by the play_game function. The requirement of using this function is the player's balance is greater than 3ETH, and he has committed his seed, which is the minimum number of ETH needed to start a game. If there is no other player is waiting, then he will be player1, change his game status to "playing", record his committed seeds to player1, and waiting other player join. If there is a player1 but no player2, he will be assigned to player2. If all 2 players have been assigned, then he need to wait them finish their game, or call timeout_for_others function to reset the game when the game is stuck for a long enough time.

1.4 Validate seeds and generate new seeds for dice roll.

This process is used to verify the committed seed and change it to a new seed according to the validated secret string, so nobody can cheat. The validate process only can start after there are 2 players in the game that they all have committed their seeds. Then these 2 players need to call validate function with an input string: their secret string that generate the seed. The contract will do abi encode, and keccak256 hash on the string and compare the result with seed, if they are equal, the validation pass. After this, the contract will merge the verified message and the original seed and apply abi encode and keccak256 hash on it to generate a new seed. The reason I do this is since if we do not do this step, the player2 can get player1's seed by reading the transaction, and by trying different values, carefully picked an input value to control the result. But after doing this, since other player has no way to know the input string before the validation process, when he able to see that string from transaction, he already committed a seed which can not be changed, in this way we can make the game fair.

1.5 Start game.

After the 2 players finished the validation, anyone of them can call the start_game function to roll the dice. It will generate a value between 1~6 and add/reduce player's balance automatically according to the generated value. After this, all data about playing game will be set to original value, and ready for the next round.

1.6 Collect reward.

When the player wants to leave and collect his reward, he can simply call the withdraw function, if he is not playing game at this moment, he can successfully let the contract send his left balance to his account.

1.7 Add balance to your account.

Sometimes the player may end up in a situation that he loses too much balance and cannot start a new game, then he can call the add_balance function to send more ETH to the contract to increase his balance in order to play more games.

My contract also has some methods to handle stuck.

In my contract, I have a Boolean variable timer, which is set to false in default, when the second player joined the game (which means game can start in theory), the timer will set to true to represent timing starts, then there are some functions that can be called by different people to reset the game when the timer starts too long. Details about how this work will be explained in vulnerability part.

Detailed answer for specific questions (most of them is answered above, here is just some summaries and additions):

Who pays for the reward of the winner?

The loser of the game pays for the reward. Each player needs to have at least 3 ETH of balance to start a game. After a game finish, the contract will increase winner's balance, and reduce loser's balance according to the game result.

How is the reward sent to the winner?

Here we use pull over push to send the reward. Which is let the user itself call the withdraw function to take back his balance. And in my implementation, the user does not need to call this function every time. He can leave the reward in his balance, and after he play many games and don't want to play more, he can call the withdraw function once in the end and take all the balance left.

How is it guaranteed that a player cannot cheat?

As I mentioned in previous section, before players join any game, they need to do preprocessing on a secret string by using abi encode and keccak256 hash to create a seed and commit it to the contract. This will be their initial player seed. And then they can join a game, after a game get 2 players that both have committed seed, they will move to validate stage, here they will send the secret string to the contract to verify their commit. The contract will apply the same preprocessing method on the secret string and check if it equal to the seed, if equal, then check passed, the contract will then merge the string with seed, and apply the same preprocessing on it again to generate a new seed. Since no one knows the player's secret string before this validation process, they can only read the original seed from transaction, they can not know what the new seed will be. So, it is impossible for them to calculate a seed that can control the dice result. When they can see the secret string from validation transaction, they already commit their seed and cannot change anymore. So we can say players cannot cheat.

What data type/structures did you use and why?

1. I use a self-defined struct: "player" to record each player's account information.
   The reason for this is:
   a) A player struct can make further data processing more convenient, and make the code looks clearer.
   b) It can store each round's reward/take each round's penalty at the balance field. In this way, the users can save some gas fee and the game system will be more user friendly.
   c) It can easily be stored using a mapping, which helps finding required information later.

2. I create an enum called game_state to store the current state of the player, it can improve the code's readability and makes data processing easier and can save storage space a little bit.

2: Gas evaluation of the implementation.

2.1 The cost of deploying and interacting.

Deploying:

the sender address is : 0x911076247b3d05260b0dD2C0C2137e2A2175dE7E
the transaction id is:
0x7e94e758e86d3436eb051f0e4ced752bf2c006bca1114e1246832522f22b3294
the transaction cost is 0.00351839BTL_ETH=3518390 gwei

Interacting:

Since there are many interacting methods, to make the work tidy, I will put all details of transaction id and sender/receiver address for these methods at the end (transaction history part).

First, let's see the main process for a new player to play a game in a normal case.
1.registration will cost about 109000 gwei.
2.Commit seed will cost about 50000 gwei.
3.Join game will take cost 150000 gwei.
4.Validate seed will cost about 42000 gwei.
5.Call start game will cost about 76500 gwei.
6.Withdraw balance will cost about 22000 gwei.

And there are some other interactable functions that not necessary for every round.
7.Function check_time that used to check if the 2 players in game uses too much time will cost about 44000 gwei.

8.Function exit_from_wait_validation that used to reset game when the opponent has not completed the validation after a long time will cost about 73000 gwei.

9.Function exit_from_wait_join that used to leave game for player1 when there is no player2 joined will cost about 33000 gwei.

10.Function add_balance that used to add balance for players will cost about 31000 gwei.

2.2 Whether the contract is fair to both players

Overall, the contract's cost is relatively fair but not completely fair. Process 1~4 is required for every player to play a game, so this part is fair, but process 5 and 6 is not necessary for everyone.

First, it's talk about process5, since my contract only need 1 player to start the dice roll, so only one player will run this function. We can see mandatory process 1~4 will cost around 207900 gwei, if add process 5's cost to it, it will become 284400, which means the player who don't pay the start game cost's total cost is 73% of the other one. Which means the caller of this function will pay 27% more. Although I can achieve fair by require both players to call the start game function, but it would lead to an unnecessary waste of resources, so I do not think this is a good idea.

The other unfair function is withdraw, but the effect of this function is much smaller, since its cost on 22000 gwei, which is much less than start_game's cost 76500. And it happens much less often, it only happens when the dice result is 3 or 6, and the loser only have 3 in balance, then he does not need to call withdraw function later since he has no balance. In this case, the winner already gets 3 eth reward, which is much higher than the cost of withdraw, so I think winner pays little bit more cost in an uncommon case is reasonable and not a big problem.

2.3 Techniques to make the contract more cost efficient and/or fair

1: I create a struct to store player's information. By doing it, I do not need to use many

variables to store the related data, which can save storage space to reduce the cost.

2: My contract gives each registered player a balance field, in this way, they do not need to withdraw the reward for every round, they can save the reward at balance and take all the balance when they want to leave in the end, in this way player only need to call withdraw function once, which can save many gasses cost.

3: I use enum to store player's game state, since solidity will automatically process enum's value as integer, by using enum instead of several strings, I make the contract more cost efficient.

3: Potential hazards and vulnerabilities, and the way of mitigating.

3.1: Griefing

Griefing is a type of denial-of-service attack, the attacker uses some ways to make a transaction with contract fails, and it may affect other operations that after the failed transaction. For example, if the contract use send() function to send money to an attacker contract, the attacker contract can make the fallback function don't have enough gas to execute, so the send will fail, which will make the smart contract's later operation unable to continue.
Here in my contract, I use a pull over push method to mitigate this problem, I let the player call withdraw function to get back his balance. In this way, each eth send call is isolated, and the gas is paid by the caller, then we can prevent griefing attack from doing that.

3.2: Reentrancy

Reentrancy attack can happen when the contract calls an external contract before finishing all state processing, the external attacker may use fallback function to let the contract do things it has did again, for example, send eth to the attacker again and again.
Here in my contract, I solved this problem by finishing all the state change before any eth transfer. For example, in withdraw function, I set the player's balance to 0 before send the withdraw eth to him, by doing it, if the player applied the reentrancy attack, no eth will be sent in the next round, which prevent the reentrancy attack.

3.3: Front-running

When an attacker saw a transaction, he can use information from the transaction to create his transaction and pay more gas on it to make miners process the transaction earlier than the original one. This is front-running attack.
But my contract does not need to worry about this problem, since in my implementation, the

order of transaction will not influence the game, so the attacker can not gain profit from doing front-running here.

3.4: The attacker may get information from earlier transaction and cheat by that.

Since all transaction can be seen by other people, the attacker may get information from previous transaction, for example, they may get the other player's committed seed from that and try to calculate their seed to control the dice result.
In my contract, I mitigate this problem by adding a validation process. The validation process can only happen when both 2 players has committed their seeds. In this process, the contract will require the player to send the secret string that used to generate the seed, and the contract will run the preprocessing by itself and compare the result with seed, if they are equal, then validation pass. Then the contract will merge the secret string with original seed and generate a new seed for later dice rolling process. Because the secret string never shown in a transaction before this process, when the attacker commits his seed, the only thing he can see is the other player's original seed, but without the secret string, he can not compute a value to control the dice result. And when he able to see the secret string, he already committed his seed which cannot be changed anymore. So, information from earlier transaction cannot help attacker cheat.

3.5: The game may get stuck.

The game may get stuck there if 1 or all 2 players do not validate their seed or call start game. It will make later players unable to play games. The way I use to mitigate this problem is using a timer. Once both 2 players called join game (which means the game can start in theory), the timer will set to true and record the current time in current_time variable. I write 2 different methods to call timeout, 1 for player in game and 1 for another players. If one player is ready to play game but the other player never validates his seed, then if the time pass the waiting time limit, the validated player can call timeout to exit the game and get the opponent's 3 eth as punishment. If both players do not do action to continue the game, there is a timeout function for third party players, he can call this function when time passed the limit to reset the game and get both player's 3eth as punishment for them. By doing this, we mitigate the game stuck problem.

4: Tradeoffs

1. As mentioned in the gas cost part, I set the game only need 1 player to call the start_game function. Because I think it's a waste of resources to have two people doing the same thing when there's no difference between do it once and twice. But there is no doubt that this makes the fairness of cost of gas decreases. It's a tradeoff between efficiency and fairness.

2. I use a push over push method to send the reward. The players need to call the withdraw function by themselves to get the reward, although it can prevent some security attack, but it will make user experience decrease since they need to call 1 more function and pay extra gas fee for it.

3. The last trade of is the commit and validation process, if we assume all players are honest, we can just use the block's information as seed to generate random number, it will make the game process faster and cost less gas. But this also will give attacker chance to cheat. So here I chose to give up part of the user experience and efficiency to improve security and fairness

5: Fellow student's contract analysis

My fellow student's contract is implemented in a different way. His process of playing game is based on a struct called game, which looks like this:

He stores most of important data about playing game on the struct: Game. His idea is let the player1 create a game, input required information to Game struct, and wait the other player join the game and input his information to Game. Then they will call the verify function separately. His hashed value for commitment is generating from integer, after verifying, the verified input integer will also be stored in "Game". And his random number is generating by hash of the 2 input integers and block number. After game finished, he also uses a pull over push method to let the player withdraw his reward.

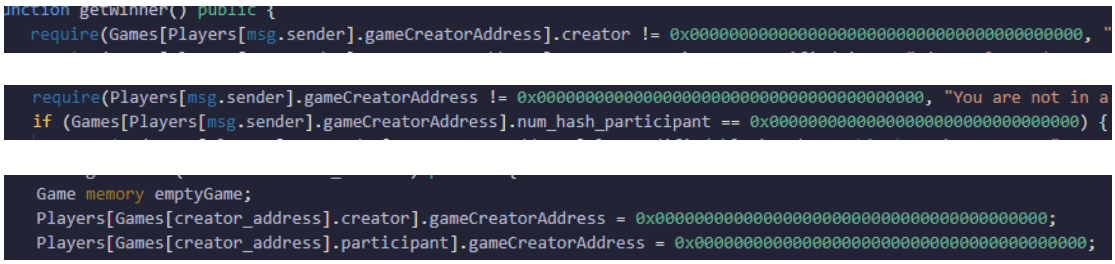
The core function of his contract is solid, but I still found some small problems that can be improved.

Firstly, in his first version of contract, he does not have a stuck handler in his contract, an attacker could exploit this vulnerability to perform a denial-of-service attack. He can join player1's game and never do the commitment verify, then player1 will be stuck here forever. He needs a handler to solve this problem, just like in my contract, I use a timer to check if the game passes the time limit, if timeout detected, the player that stuck here should be able to call a function to reset the game. After I told him this problem, he added stuck handler in his later version of contract.

And he also sets that the final roll dice function only needs to be called by one player, it will

make his contract not completely fair since one player will pay cost more gas for one round of game.

Besides that, for some specific value that very complex but often be used, I think it's better to store it in a private variable, which can largely increase the readability of the code. For example,



The special value 0x00000000000000000000000000000000 is often be used in his contract, and this value is very long, add those to the code line will make it hard to read. It will be good to assign it to a byte32 variable called empty_value to make the code tidier.

6: The transaction history of an execution of a game

Deploy contract
Sender: 0x7E8Bd6114Cd35e1C1DbE681AEA97429F9fa480da
Transaction ID:
0xfdf47f24849908217cde0f64acdafe2696c4a68214ebd3347f2804caf6b65bb7

Player1 Registration
Sender: 0x7E8Bd6114Cd35e1C1DbE681AEA97429F9fa480da
Receiver: 0xAA184470Da7f3D71ed6ad42E0D5B6e7B2d8cD628
Transaction ID:
0x9737ec240b617a0e3b5c3bced3fe66801b6b03e2fc72192c4515f16f399287b2

Player2 Registration
Sender: 0x911076247b3d05260b0dD2C0C2137e2A2175dE7E
Receiver: 0xAA184470Da7f3D71ed6ad42E0D5B6e7B2d8cD628
Transaction ID:
0x0ba177530e9392c4f3063f632bf80bed6c0a01add174f42049b74a9703c75f5b

Player1 Commit
Sender: 0x7E8Bd6114Cd35e1C1DbE681AEA97429F9fa480da
Receiver: 0xAA184470Da7f3D71ed6ad42E0D5B6e7B2d8cD628
Transaction ID:
0x265200fae3595b45f1a259acdd102b57e09d4800506dde406ebd5ba7f4aac3e6

Player2 Commit
Sender: 0x911076247b3d05260b0dD2C0C2137e2A2175dE7E
Receiver: 0xAA184470Da7f3D71ed6ad42E0D5B6e7B2d8cD628
Transaction ID:
0x9b5bb7fe34d56f050e12aab0f4ff92ea43bc4a18cc254cab80c5f199b5e44404

Player1 Join game
Sender: 0x7E8Bd6114Cd35e1C1DbE681AEA97429F9fa480da
Receiver: 0xAA184470Da7f3D71ed6ad42E0D5B6e7B2d8cD628
Transaction ID:
0x10a988c7491008429179f79a6e3fea2f373829914be4e5210c9f49bea7f23a1d

Player2 Join game
Sender: 0x911076247b3d05260b0dD2C0C2137e2A2175dE7E
Receiver: 0xAA184470Da7f3D71ed6ad42E0D5B6e7B2d8cD628
Transaction ID:
0x11a030c05e60c5d5f628e2221d1c4d6b08cac26de552fd1aeba3fc1c3a14ff4b

Player1 validate seed
Sender: 0x7E8Bd6114Cd35e1C1DbE681AEA97429F9fa480da
Receiver: 0xAA184470Da7f3D71ed6ad42E0D5B6e7B2d8cD628
Transaction ID:
0x6bf3ffe54ab88480cf7134a99a71229e4db631104632f96f6a4570802459666e

Player2 validate seed
Sender: 0x911076247b3d05260b0dD2C0C2137e2A2175dE7E
Receiver: 0xAA184470Da7f3D71ed6ad42E0D5B6e7B2d8cD628
Transaction ID:
0x62cf402d6e46fc31ba9e9cc3639f7b4a449ca41bf75aa0db89982d9d6dea81da

Player1 start game
Sender: 0x7E8Bd6114Cd35e1C1DbE681AEA97429F9fa480da
Receiver: 0xAA184470Da7f3D71ed6ad42E0D5B6e7B2d8cD628
Transaction ID:
0xfa4d3b1a813db2de9afc17fc241d14388e76cf4b7f675e78b77bc50fb112fb89

The dice result is 2: 

Player1 withdraw balance
Sender: 0x7E8Bd6114Cd35e1C1DbE681AEA97429F9fa480da

Receiver: 0xAA184470Da7f3D71ed6ad42E0D5B6e7B2d8cD628
Transaction ID:
0x23ef7fc70e2312927d20584fb3c3ca34a7bc3ba51354ecd84f722599d9fd4538

Player2 withdraw balance
Sender: 0x911076247b3d05260b0dD2C0C2137e2A2175dE7E
Receiver: 0xAA184470Da7f3D71ed6ad42E0D5B6e7B2d8cD628
Transaction ID:
0x1206bea18137e4e9922e1ba50dcb8d3ce33e20d4f4422793d7bc4653ecb8f986

7: code of my contract

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract s1935167 {
    address owner;

    bool public timer = false;
    bool public time_passed = false;
    uint private start_Time = block.timestamp;
    uint private current_time;
    uint private time_Limit = 1 minutes;


    struct player {
        address player_address;
        uint balance;
        bool exist;
        bytes32 seed;
        game_state state;
        bool validated;
    }

    event Withdrawal(address customer);
    event Add_balance(address customer);
    player public player1;
    player public player2;
    player null_player = player1;
    mapping(address => player )public players;
```

```
    enum game_state{
        free,
        commited,
        playing
    }
    uint public result;

    function check_time()public returns(bool){
        current_time =  block.timestamp;
        if (current_time-start_Time > time_limit){
            time_passed = true;
            return true;
        }
        return false;
    }


    /*
    This method can let player to do registration, and store at least enough balance for 1 round of game.
    */
    function register() public payable returns (string memory) {
        require(msg.value >=300000000000000000 && !players[msg.sender].exist,"you should not registered before or you at least should send 3ETH to regist");
        player memory p = player(msg.sender,msg.value,true,keccak256(abi.encodePacked(msg.sender)),game_state.free,false);
        players[msg.sender]  = p;
        return "you are successfully registed";
    }
```

```
    function add_balance() public payable returns(uint){
        require (players[msg.sender].exist && msg.value > 0,
            "you need to be a registered player, and send value >0");
        players[msg.sender].balance += msg.value;
        emit Add_balance(msg.sender);
        return players[msg.sender].balance;
    }

    function commit_seed(bytes32 new_seed) public{
        require(players[msg.sender].exist && players[msg.sender].state == game_state.free);
        players[msg.sender].seed = new_seed;
        players[msg.sender].state = game_state.commited;
    }

    function join_game() public{
        require(check_balance(players[msg.sender].player_address) && players[msg.sender].state == game_state.commited && (!player1.exist || !player2.exist),
        "you need to commit first or wait the last game finish or store enough balance");
        if(player1.exist){
            player2 = players[msg.sender];
            players[msg.sender].state = game_state.playing;
            timer = true;
            start_Time =  block.timestamp;
        }
```

```
        else{
            player1 = players[msg.sender];
            players[msg.sender].state = game_state.playing;
        }
    }

    function validate(string memory message)public returns (bool){
        require (player1.exist && player2.exist && (players[msg.sender].player_address == player1.player_address
        || players[msg.sender].player_address == player2.player_address),"validation only for playing players, and the game should have 2 players joined");
        if (players[msg.sender].player_address == player1.player_address){
            if (keccak256(abi.encode(message))==player1.seed){
                player1.validated = true;
                player1.seed = (keccak256(abi.encodePacked(player1.seed,message)));
                return true;
            }
            else{
                return false;
            }
        }
        else{
            if (keccak256(abi.encode(message))==player2.seed){
                player2.validated = true;
                player2.seed = (keccak256(abi.encodePacked(player2.seed,message)));
                return true;
            }
            else{
                return false;
            }
        }
    }

    function start_game()public{
        require(player1.validated && player2.validated,"both players need to be validated");
        result = uint(keccak256(abi.encodePacked(block.timestamp, msg.sender, block.number,player1.seed,player2.seed))) % (6);
        result += 1;
        players[player1.player_address].state = game_state.free;
        players[player2.player_address].state = game_state.free;
        if (result>3){
            result -= 3;
            players[player1.player_address].balance -= result*100000000000000000;
            players[player2.player_address].balance += result*100000000000000000;
        }
        else{
            players[player2.player_address].balance -= result*100000000000000000;
            players[player1.player_address].balance += result*100000000000000000;
        }
        player1 = null_player;
        player2 = null_player;
        timer = false;
    }

    function players_ready() public view returns (bool){
        require(player1.exist && player2.exist);
        return true;
    }

    /*
    used when there is no player2 joined and player 1 want to leave
    */
    function exit_from_wait_join()public{
        require(players[msg.sender].exist && player1.player_address == msg.sender && !player2.exist);
        players[player1.player_address].state = game_state.free;
        player1 = null_player;
```

```
    }

    /*
    used when the other player not validate for a long time
    */
    function exit_from_wait_validation()public{
        require(timer && players[msg.sender].exist && (player1.player_address == msg.sender || player2.player_address == msg.sender));
        if (check_time()){
            if (player1.player_address == msg.sender && !player2.validated && player1.validated){
                players[player2.player_address].balance -= 3*100000000000000000;
                players[player1.player_address].balance += 3*100000000000000000;
                players[player1.player_address].state = game_state.free;
                players[player2.player_address].state = game_state.free;
                player1 = null_player;
                player2 = null_player;
                timer = false;
            }
            if (player2.player_address == msg.sender && !player1.validated && player2.validated){
                players[player2.player_address].balance += 3*100000000000000000;
                players[player1.player_address].balance -= 3*100000000000000000;
                players[player1.player_address].state = game_state.free;
                players[player2.player_address].state = game_state.free;
                player1 = null_player;
                player2 = null_player;
                timer = false;
            }

        }
        players[msg.sender].state = game_state.free;
        player1.player_address = address(0);
        players[player1.player_address].exist = false;
    }

    function timeout_for_others() public{
        require(timer && players[msg.sender].exist && player1.player_address != msg.sender && player2.player_address != msg.sender);
        if (check_time()){
            players[player2.player_address].balance -= 3*100000000000000000;
            players[player1.player_address].balance -= 3*100000000000000000;
            players[msg.sender].balance += 6*100000000000000000;
            players[player1.player_address].state = game_state.free;
            players[player2.player_address].state = game_state.free;
            player1 = null_player;
            player2 = null_player;
            timer = false;
        }
    }



    function check_balance(address player_address)public view returns(bool){
        require(players[player_address].exist);
        if (players[player_address].balance >=3*100000000000000000){
            return true;
        }
        else{
            return false;
        }
    }

    function withdraw() public {
        require(players[msg.sender].state != game_state.playing);
        uint b = players[msg.sender].balance;
        players[msg.sender].balance = 0;
        payable(msg.sender).transfer(b);
        emit Withdrawal(msg.sender);
    }
```