Blockchain and Distributed Ledger – Assignment 2

S2300928

Contract system design

To keep a stable contract balance, a “buy in” model is used. The players each need to buy their way into a game round by paying 1.5 ether each. This ensures that the highest reward is technically “crowd-funded”. Thus, the contract has a 33% chance to stay in profit, and 66% chance to not lose its balance value. The reward is added to the user balance, where the user can pull their value in a withdrawal, as desired.
This gives a chance for the contract to accumulate balance, and eventually be able to “bankroll” rewards for the said players, much alike a real-life casino.

To prevent the possibilities of cheating, overflow and underflow checks are implemented before any player joins or creates a room. It is also ensured that a player cannot play the game with themselves, as this would give them the same chances of not losing money as the contract. The users are asked to commit a Keccak256 hash of a secret message each. Upon both the players joining the room and providing the commit, the players reveal their commits by sending the actual secret message, that are hashed to verify secret. It is important to note that none of the previously used committed hashes can be committed again, as this makes the contract vulnerable to preimage attacks by attackers. A commitment from the contract is used which is a hash of a future block. These three commitments are XOR’d and the is hashed, and mod 6 +1 value is taken as the outcome of the roll of the dice.

The data types and structures used:

1. Struct: to define the “Game”, which essentially stores details corresponding to the game
2. Enum: to define game states
3. Map: to provide a map of user address and balance, used to retrieve and update player balances, and to provide a map of commits, and whether they have been used or not
4. Array: to store player addresses, and to store all committed hashes as keys for the map
5. String array: used for several variable parameters
6. Bytes32: used to store Keccak256 hash
7. Addresses
8. Primitives: these include uint256 and Boolean

The normal control flow of the program:

1. The program is deployed by the developer.
2. Developer runs init(). This sets the first address in player array to be that of the developer. This allows them to empty the contract balance into their account.
3. Join room is called by player 1 with keccak256 output of their secret and at least more than 1 ETH (>1.5e18 + 5e4 wei fee for buy in and half of gas for play() method)
4. Join room is called by player 2 with keccak256 output of their secret and similar amount
5. Player 1 calls getBalance() to check their balance in the game state after the buy in is deducted
6. Player 2 calls getBalance() to check their balance in the game state after the buy in is deducted
7. Player 1 reveals their secret by calling playerReady() and passing their secret, which is verified by contract by hashing it with keccak256 algorithm and equating with the previously committed value.
8. Player 2 waits for 2 more blocks to be mined after the current, and reveals their secret by calling playerReady() and passing their secret, which is verified by contract by hashing it with keccak256 algorithm and equating with the previously committed value.
9. Player 1 waits for 2 more blocks to be mined after the current and then calls play().
10. The contract XORs the committed secrets and the current block's hash and uses it to generate dice roll outcome. Depending on that, balance is added to the winner according to the given constraints.
11. Player 1 calls getBalance() to check their balance, to see if they won, and withdraw balance if they choose to.
12. Same step is repeated by Player 2.
13. This flow is repeated for as many times as desired.
14. Once the developer has made enough money, they call the empty() function, which returns all the ETH the contract has into their address, and the contract self-destructs.

Other functions:

1. freeGame(): Frees a game if stuck
2. getGameState(): Used to check game state (value of enum State)
3. willSatisfyBlockOffset(): Checks if enough blocks have passed for next operation

Evaluation of implementation:

The following approximates gas costs in different scenarios in the game. GAS PROFILER (https://github.com/EdsonAlcala/remix-gas-profiler) has been used to estimate the following values:

Method Name	Scenario	Cost (gas units)
Deployment	Any	1552565
joinRoom()	Creating lobby, first time playing	209234
	Creating lobby, has balance	163596
	Joining lobby, first time playing	176663
	Joining lobby, has balance	130239
Init()	Any	63052
playerReady()	First time call	80079
	Second time call	63600
play()		45402
withdraw()		29800
getBalance()	Called by Contract	23591
	Other	0
freeGame()		47780
getGameState()	Called by Contract	23754
	Other	0
willSatisfyBlockOffset()	Called by Contract	23518
	Other	0
empty()		30645

The contract is fair. Both the players pay equal buy-in, which covers the maximum reward of the game. It may seem from the table that the caller of play() function would need to pay entirely for the gas for the method call, which is vital as that is the point where the dice roll is calculated. This is avoided by collecting estimated gas fees from both the players at the time they call joinRoom(), which is sent to the caller of the play() method.
To make it cost efficient, players directly send ETH while calling joinRoom(). This saves gas for adding a separate deposit method. For both fairness and cost-efficiency, a pull method of withdrawal of player balances is used. This prevents the scenario where the player who calls the play() method also pays for the transfer of funds to the winner. Also, this improves the security of our contract.
For further fairness and ensure pseudo randomness, a commit and reveal scheme is used and a future block's hash to do an XOR on all three values as discussed before, for calculating the result of the dice. If an attacker has 51% control of a network's hash however, they may alter contents of the block storing the game variables and may alter the network in a way (by declaring blocks with precision) that they control the block hash of the future block that is used in our pseudo randomness generation algorithm. However, since these attacks can happen to any blockchain, and it is impractical for one entity to hold that much hash-rate in any current major blockchain network (cost Vs reward is very poor, especially if done for this game), it can be considered improbable and ignored.

Also, to be practical in terms of the contract balance, as discussed above, a buy in method is used, which ensures that the contract can remain functional and serviceable to future players.

List of potential hazards and mechanisms to mitigate:

1. Denial of Service:
   a. A pull mode of retrieval of balances is used, instead of pushing the winnings to the players.
   b. No unbounded loops used, or any loops for that matter.
2. Re-entrancy:
   a. Balance from a user’s balance map is removed before transfer (mutex-like implementation)
   b. A pull mode of withdrawal is used instead of push
   c. payable.transfer() is used instead of call()
3. Preimage attack: It is ensured that none of the commits used in current game have been used before. This prevents attackers from exploring the blocks and keeping track of hashes and revealed secrets and checking if the same hash is committed again, to launch a preimage attack based on this.
4. Miscellaneous:
   a. Forcibly sending ETH to contract: this.balance not used
   b. Delegate call exploit: <<library>>.delegatecall() not used
   c. Tx.origin exploit: tx.origin not used
5. Default value exploit (e.g., Nomad-Bridge Hack) and control flow distruption: Default value conditions are checked via game states and Booleans used as mutexes
6. Front-running: A commit-reveal scheme is used separated by block numbers (as in, for the other player to commit, they must wait for further two blocks to be mined, and same to call the play() method which calculates the roll of the dice
7. Overflow/Underflow exploit: Solidity version >=0.8.0 is used, which has an overflow/underflow check, and can check if the values will overflow or underflow after playing a game, in the best and worst outcomes for a player. This prevents the game state to fail/revert after balance has already been deducted.
8. Pseudo-randomness: To ensure true randomness as far as the players are concerned, a pseudo randomness number generator scheme is implemented where the two players commit a Keccak256 hash of their chosen secret, and after one of the players confirms their secret (verified by generating Keccak256 hash of the secret and equating with the previously submitted hash of the player), the next one can confirm theirs only after at least 2 more blocks have been mined on the chain, and one of them can only call the play() method after at least a further 2 blocks gave been mined. The current block’s hash is used to XOR with the secrets committed by the two players.

Trade-offs and choices made between security/performance and fairness/efficiency:

1. A pull method of withdrawal has been used, instead of push, as mentioned before. This may sightly hamper the user experience, but greatly improves the security of our contract.
2. No previously used committed hashes are allowed to be used again, to prevent preimage attacks. This hampers UX but increases security of the contract.
3. Due to security reasons in terms of randomness, the play() method runs at least after 2 blocks from the block where both the players have “revealed” their secrets. This causes one of the users to end up paying more. For this, both the players pay part of the estimated gas fee during the joinRoom() call, and whoever calls the play() method, gets their balance updated with the estimated gas fee in credit.
4. Players need to call the contract at least twice or thrice, which means more gas to be paid by them. But this does greatly increase the chance of assuring an anti-cheat scheme, so none of the players can be swindled by the other.
5. When the committed secrets are XOR’d with the current block hash when the play() method is called, the only threat could be a 51% attack, by which the attacker can control the block’s hash and thus practically choose the outcome. However, in this case the cost vs. reward would be too bad for an attacker to attempt this on any major blockchain. So, it has been ignored.

Analysis of partner’s contract:

The following consists of the analysis of my partner’s contract with snippets where required, to discuss the potential vulnerabilities found and how these can be exploited or cause disruption in the system.

1. Pre-image attack:

Since it is not checked that a previously used commit cannot be used again, and also no future block-based hash is included, an attacker can easily monitor the blocks mined and if any hash has been reused, he can do a pre-image attack as the blocks can giveaway both the committed hash and the secret. There is also the possibility of him having a list of preimages for keccak256, where if any matches, since no further XOR logic is present, he can choose a secret that will result in the decision being in their favour.

```
function commitHash(bytes32 commitment) public {
        require(balance[msg.sender] >= MAX_ETHER + FEE);
        State state_copy = state;
        if (state_copy == State.FirstCommit) {
            state = State.SecondCommit;
            player1Storage = commitment;
            player1 = msg.sender;
        } else if (state_copy == State.SecondCommit) {
            state = State.FirstReveal;
            player2Storage = commitment;
            player2 = msg.sender;
            revealDeadline = block.number + REVEAL_SPAN;
        } else {
            revert();
        }
        balance[msg.sender] -= FEE;
    }
```

```
function calculateResult() public {
        State state_copy = state;
        require(state_copy == State.ResultAwait || block.number > revealDeadline);
        uint val;
        if (state_copy == State.FirstReveal) {
            // The first player did not reveal value, therefore ether goes to the
second player
            val = 6;
        }
        else if (state_copy == State.SecondReveal) {
            // The second player did not reveal value, therefore ether goes to the
first player
            val = 3;
        }
        else if (state_copy == State.ResultAwait) {
            val = uint(player1Storage ^ player2Storage) % 6 + 1;
        } else {
            // According to the current status, it is not yet time to count the
results
            revert();
        }
```

2. Overflow issue:

Overflow isn't handled in the possibility that upon adding of the ETH to the balance which may increase to a value over 2^256-1. Since Solidity version range is between 0.8.0 and 0.9.0, this would cause the method to fail.

```
        if (val<=3) {
            balance[player1] += val * (1 ether);
            balance[player2] -= val * (1 ether);
        } else {
            balance[player1] -= (val - 3) * (1 ether);
            balance[player2] += (val - 3) * (1 ether);
        }
```

Since withdrawal requires the condition that the caller isn't one of the players, this would call the game state to get stuck.

```
function withdraw() public {
        require(msg.sender != player1 && msg.sender != player2);
        uint256 b = balance[msg.sender];
        balance[msg.sender] = 0;
        payable(msg.sender).transfer(b);
    }
```

Transaction history of one run of the game:

Actors involved: A1 and A2

Progression of the game:

Step 1 – A1 deploys contract:

a. Gas: 1645444 units
b. TransactionId: 0xe0d89bcb83ce822fcc6ec693f410f8f0d637276ee2baae6d7081fd0caf7c2522
c. Contract Address: 0x0D9FB35652Ea4D7DB264Eeeb30af1689335De512

Step 2 – A1 runs init():

a. Gas: 63028 units
b. TransactionId: 0xacdb3969dadc4e1ec05da06e144d0f82cb4e491ce3607bb245b5d3e96ddab550

Step 3 – A1 calls joinGame():

a. Gas: 208159 units
b. Committed Hash: 0xd43a4cbc88b0a7b504d0d78238eaca085f16f41f8ac80918c69545a3c439aa0e
c. Paid: 3 ETH
d. TransactionId: 0x8a3d81178015b3b13bf35fc8c671194ec7f7546a135a89adf37c544c7fc32aec

Step 4 – A1 calls getBalance():

a. Value returned: 1499999999999975000 (wei) (1.5 ETH + 5e4 wei taken as fee)

Step 5 – A2 calls joinGame():

a. Gas: 177100units
b. Committed Hash: 0xcce4b98d9f8ef7e98dcd71afe0925beaa42c2bf3ca0853d9b3054008be28e176
c. Paid: 3 ETH
d. TransactionId: 0x37bf9757dee6c76313f69d1cdb0288bad0fc8e874eb5bc4e5f1857e6c72c0d6d

Step 6 – A2 calls getBalance():

a. Value returned: 1499999999999975000 (wei) (1.5 ETH + 5e4 wei taken as fee)

Step 7 – A1 calls playerReady():

a. Gas: 75753 units
b. Revealed Secret: specialWORD1
c. TransactionId: 0xc989e84cb9d60ae8926e35405c87b2338462f808852d07e943b4c0b3a589e66c

Step 8 – A2 calls playerReady():

a. Gas: 62564 units
b. Revealed Secret: sec14Lw0RDtW0
c. TransactionId: 0xf2e589ed1bce8b7ed52f730ee0d6abc288df0f4af9431f07add1b5901c60b230

Step 9 – A1 calls play():

   a. Gas: 46412 units (Got refurbished 50000 wei in balance)
   b. TransactionId: 0xb17411592f0847e61bfdc35ca07fcba8441a6be03eb08619db54e07a6f796628

Step 10 – A1 calls getBalance():

   a. Value returned: 3500000000000025000 (wei)

Step 11 – A2 calls getBalance():

   a. Value returned: 1499999999999975000 (wei)

(RESULT: A1 won 2 ETH, dice rolled 2, contract kept 1 ETH)

Step 12 – A1 calls withdraw():

   a. Gas: 21000 units
   b. TransactionId: 0x5ddb17c36076f13eb8223fbf981b49ae16b5e34e5b922e807ad346de43b1a91f
   c. Value received: 3.5 ETH

Step 13 – A2 calls withdraw():

   a. Gas: 21000 units
   b. TransactionId: 0xe1174f8964c8d0b7442c141ad00206ddb4199d8d495962940ca98e48f65e687c
   c. Value received: 1.4999 ETH

Step 14 – A1 calls empty():

   a. Gas: 14023 units
   b. TransactionId: 0x56ebb9409765d072117d333cf4b250a6e17e922a2ea36010119a6a818f4b870d
   c. Value received: 1 ETH

Code for the game:

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.8.0 <0.9.0;

contract s2300928 {
    struct Game {
        address address1;
        bytes32 p1commit;
        string p1secret;
        address address2;
        bytes32 p2commit;
        string p2secret;
        bool p1Ready;
        bool p2Ready;
        State state;
        uint256 lastCommitBlockNumberWithOffset;
    }

    enum State {
        INACTIVE,
        IN_LOBBY,
        LIVE
    }

    Game private game;
    mapping(address => uint256) balance;
    address[] private players;
    bytes32[] private commitedHashes;
    mapping(bytes32 => bool) commitedHashesMap;

    event Deposit(address customer, uint256 amount);
    event Withdrawal(address customer);
    event DeclareResult(address customer, string message);

    uint256 BLOCK_OFFSET = 2;
    uint256 PLAY_FEE = 50000;
    uint256 BUY_IN_FEE = 1.5e18;

    function checkIntegerOverflowAndUnderflow(
        uint256 a,
        uint256 b,
        uint256 c,
        uint256 d
    ) private pure {
        require(a + b > c); //checks integer underflow for playing, and overflow
for depositing balance
        require(b + a + d - c > 0); // checks integer overflow for winning
    }
    function withdraw() public {
        uint256 b = balance[msg.sender];
        balance[msg.sender] = 0;
        payable(msg.sender).transfer(b);
        emit Withdrawal(msg.sender);
    }

    function getBalance() public view returns (uint256) {
        return balance[msg.sender];
    }

    function willSatisfyBlockOffset() public view returns (bool) {
        return game.lastCommitBlockNumberWithOffset < block.number;
    }

    function rollDice() private view returns (uint256) {
        bytes32 p1Rand = keccak256(abi.encodePacked(game.p1secret));
        bytes32 p2Rand = keccak256(abi.encodePacked(game.p2secret));
        bytes32 contractRand = keccak256(
            abi.encodePacked(blockhash(block.number))
        );
        return
            (uint256(
                keccak256(abi.encodePacked(contractRand ^ p1Rand ^ p2Rand))
            ) % 6) + 1;
    }

    function play() public {
        require(willSatisfyBlockOffset() && game.p1Ready && game.p2Ready);
        uint256 result = rollDice();
        if (result < 4) {
            balance[game.address1] += (result) * 1e18;
            emit DeclareResult(game.address1, "Won!");
        } else {
            balance[game.address2] += (result - 3) * 1e18;
            emit DeclareResult(game.address2, "Won!");
        }
        game.state = State.INACTIVE;
        balance[msg.sender] += PLAY_FEE;
    }

    function freeGame() public {
        require(game.state == State.LIVE);
        balance[game.address1] += 1e18 + PLAY_FEE / 2;
        balance[game.address2] += 1e18 + PLAY_FEE / 2;
        game.state = State.INACTIVE;
    }

    function getGameState() public view returns (State) {
        return game.state;
    }
    function playerReady(string memory message) public {
        require(game.state == State.LIVE);
        if (!game.p1Ready && !game.p2Ready) {
            if (msg.sender == game.address1) {
                if (keccak256(abi.encodePacked(message)) == game.p1commit) {
                    game.p1Ready = true;
                    game.p1secret = message;
                }
            } else if (msg.sender == game.address2) {
                if (keccak256(abi.encodePacked(message)) == game.p2commit) {
                    game.p2Ready = true;
                    game.p2secret = message;
                }
            }
            game.lastCommitBlockNumberWithOffset = block.number + BLOCK_OFFSET;
        } else if (willSatisfyBlockOffset()) {
            if (game.p1Ready && !game.p2Ready) {
                if (msg.sender == game.address2) {
                    if (keccak256(abi.encodePacked(message)) == game.p2commit) {
                        game.p2Ready = true;
                        game.p2secret = message;
                    }
                }
            } else if (!game.p1Ready && game.p2Ready) {
                if (msg.sender == game.address1) {
                    if (keccak256(abi.encodePacked(message)) == game.p1commit) {
                        game.p1Ready = true;
                        game.p1secret = message;
                    }
                }
            }
            game.lastCommitBlockNumberWithOffset = block.number + BLOCK_OFFSET;
        } else {
            revert();
        }
    }

    function init() public payable {
        players.push(msg.sender);
    }

    function empty() public {
        require(msg.sender == players[0]);
        selfdestruct(payable(msg.sender));
    }
    function joinGame(bytes32 commit) public payable {
        require(game.state == State.IN_LOBBY || game.state == State.INACTIVE);
        require(!commitedHashesMap[commit]); //checks that commit values aren't
used twice (otherwise attacker can know what the hash's preimage is
        checkIntegerOverflowAndUnderflow(
            balance[msg.sender],
            msg.value,
            BUY_IN_FEE + PLAY_FEE / 2,
            3e18
        ); //1.5 eth entry fee + gas for calling play() function, also checks for
overflow condition in event of winning 3 eth
        balance[msg.sender] =
            balance[msg.sender] +
            msg.value -
            (BUY_IN_FEE + (PLAY_FEE / 2)); //1.5 eth entry fee + some more to be
returned to user calling play()
        players.push(msg.sender);
        emit Deposit(
            msg.sender,
            msg.value - (BUY_IN_FEE + (PLAY_FEE / 2))
        );
        if (game.state == State.IN_LOBBY && game.address1 != msg.sender) {
            game.address2 = msg.sender;
            game.p2commit = commit;
            game.state = State.LIVE;
            commitedHashes.push(commit);
            commitedHashesMap[commit] = true;
        } else if (game.state == State.INACTIVE) {
            game.address1 = msg.sender;
            game.p1commit = commit;
            game.state = State.IN_LOBBY;
            commitedHashes.push(commit);
            commitedHashesMap[commit] = true;
        } else {
            revert();
        }
    }
}
```