BDL - Coursework 2

Tianjiao Wang, s1912558

November 7, 2022

1 A detailed description of the high-level decisions you made for the design of your contract, including (but not limited to):

Before answering questions, first let me show how my smart contract works.

According to the game rule, there are two players, if the dice number roll from 1 to 3 then the first player wins, if the number is 4 to 6 then the second player is the winner. The winner will win the reward ether according to the number rolled from the dice. If the first player is the winner, the reward number will be the same as the number of dice. If the second player is the winner, the reward number will be the number of dice minus 3.

Game rule In this smart contract, there are four stages: join the game with commit, reveal the game with salt and key, play the game when both players are revealed, and withdraw the reward. Each player needs to send 3 ether to join the game. So the total balance will be 6 ether. When players want to play the game, for the first player who joins the game, his/her winning condition is the dice roll from 1 to 3. So he/she will get the fee paid when joining the game (that 3 ether), and the reward (dice number). And for the second player who joins the game, the winning condition will be a dice roll from 4 to 6 and he/she will receive the fee paid when entering the game and the reward (dice number - 3) as well. The game must be played within 1 hour, and count from the first player enters the game. If the second player enters within 1 hour, the timeout will be reset; if not, the first player is allowed to withdraw the ether back and the game reset.

Deposit and balance The only method to deposit the contract is to join the game. 3 ether is the only amount of fee, lower or larger than 3 ether won't let the player deposit because lower than 3 ether won't be able to pay them enough reward and greater than 3 ether is unnecessary. The function joinGame() checks fee using the modifier isFeeCorrect() to ensure this. Before the player joins the game and pays the fee, the function joinGame() uses the modifier isJoinable() to see if there is space for the player. The number of the player must be 2, if there are already 2 players in the game, the third player will be refused when he/she tried to join and pay the fee; if the player is already in the game, he/she will be refused to rejoin the game and pay fee twice. Eventually, every player pays 3 ether to join the game, so the total balance will be 6 ether.

Join and reveal When the first player joins the game, an expired time is calculated using block.timestamp plus 3600 (1 hour). When the second player joins the game, timeout is checked to ensure the game is not

expired. If not the expired time is updated by the current block.timestamp plus 3600, otherwise second cannot join the game. The player will commit a bytes32 number, which is the hash commitment to check if the player is honest in the later game [3] [1]. When both players join the game (ensure by modifier playReady(), if there is not enough player an exception stating will be thrown) they both need to reveal the game by committing a uint key and bytes32 salt in function gameReveal() [3] [1]. The key and salt is the hash parameter that user uses to get the hash commitment when they join the game. The function will check if the player commits the correct key and salt using keccak256(abi.encodePacked(key,salt)) to see if the player is honest and if it is called by the correct player since no one except this player would know the original string including the key and salt. If they are not then the game will not start. If the player does not reveal the game until timeout, he/she will receive a punishment. The player who follows the rules and reveals the game could withdraw all 6 ether put in the game and the not reveal player could not withdraw anything. If none of them reveals the game they both won't withdraw anything.

Roll the dice When there are two players (checked by modifier playReady()) and they all reveal the game (checked by modifier isReveal()), the game could start. Function playGame() is called by one of the players to roll the dice and distribute the reward. The dice number depends on the key entered by two players, it XORs the numbers to get the seed, the seed is encode and hash again to get the final random number between 1 and 6. So there will be six combinations the result could be:

RollNumber	First player's reward	Second player's reward
1	4	2
2	5	1
3	6	0
4	2	4
5	1	5
6	0	6

And the reward each player should have is calculated in the function and distributed to each player's struct. The game is closed after playing once, repeat roll dice is not allowed.

Withdraw Only player of the game could call the function withdrawFees(). The game have punishment on timeout and not reveal, so the function is not only be called after the dice is rolled, so there will be different situations to determine the amount of ether the player could withdraw:

Timeout	Situations	Caller's withdraw (ether)	Reset game
Yes	1.Only one player in the game, allow the player quit and take ether back	3	Yes
Yes	2.The caller reveal the game but the other player didn't	6	Yes
Yes	3.Whatever players reveal or not	0	Yes
No	4.Dice rolled, the caller withdraw after the other player	reward	Yes
No	5.Dice rolled,, the caller withdraw before the other player	reward	No
-	6.Caller didn't reveal the game	0	No

[LAST SITUATION]

From the first 2 situations, when timeout happens, only when the other player didn't reveal the caller could have 6 ether, if only one player is in the game he/she will receive 3 ether back. In case 3, if timeout, whatever the players reveal or not they won't withdraw anything and the game reset, that is

the timeout punishment. Remember in the table of the situation, each situation has a sequence, if the previous situation has happened, the withdrawFee() function finishes the job. In these three cases, the caller is enough to determine whether the game is over, so after the withdrawal, the game will be reset. For situations 4 and 5 that are not timeout and dice rolled, means all players are revealed, and the caller will get their reward, if the caller is the last player to withdraw the game will be reset, otherwise, it should wait for the other player to call. The last situation is except for all situations above, whether timeout or not, the caller didn't reveal the game, in this case, nothing will happen, no ether is withdrawn and the game will not be reset, only the caller pay for the gas fee.

Reset game restGame() is a private function, it is called when withdrawal happens. It will first reset the caller's data (address, commit, key, reveal, reward) because he/she already withdraw the ether. If all player(s) in the game have withdrawn the ether, this function will reset the other parameter, including both player's withdrawal, the address list storing the player, the dice number and other boolean parameters which push the game process.

1.1 Who pays for the reward of the winner?

In the case when the dice are rolled and there is no punishment there will be the winner of the game. The reward is paid by the player who loses the game, he/she will withdraw the entering fee (3 ether) minus the reward and the winner will withdraw entering plus the reward.

1.2 How is the reward sent to the winner?

Under the case when dice rolled and have a winner. Only when the dice is rolled and calculated the reward after the game is closed the winner is able to withdraw the reward. In order to defend against reentrancy attacks, this function was built as a pull rather than a push function (See Section 3 for security). The reward of the winner is transferred to the winning address when this function is called. It is not possible to run the withdraw functions twice because the balance is reset to 0 and the winner's data including address and reward amount are all reset as well.

1.3 How is it guaranteed that a player cannot cheat?

Commit, key and salt When player enter the game a hash commit is required, that is the hash result of key and salt. This ensure the player is honest. Only when reveal success the game could start, otherwise the dishonest player will be punished, all 6 ether will be sent to the honest player.

XOR The dice number is produced by the key each player submit, and XOR the keys to get a third number, that number will generate the dice number from 1 to 6. There reason to use XOR is because the result will be depend on both players, and before user join the game there is no way for them to know other player's key. That is harder to break than a regular on-chain random number generator.

Timeout Timeout ensures the game could end at a certain time. When the first player enters the game, everyone could see the commit, some attackers may try to decode the hash commitment and join the game on purpose. Timeout in this game here limits the time for the attack to decode, it is difficult to

find the key that the first player uses within 1 hour. If the second player decodes successful but timeout, he/she cannot join the game.

1.4 What data type/structures did you use and why?

struct struct Player is defined with the address of the player, bytes32 commit, uint key, uint reward, bool reveal and bool withdraw. Other variables have been introduced, reveal is to indicate whether a player has revealed or not, and withdraw is to indicate whether a player is withdrawn or not. These two booleans will help to determine if the next step of the game should be taken.

mapping In GamePlayers key is the address, value is the struct Player. In balance, the key is address, value is the balance of the player. GamePlayers could be used to access the struct type inside Player and balance is used to store the joining fee.

array Array Players type address length 2, distinguish who is the first player and who is the second.

uint There are three uint numbers, RollNumber stores the random number of the dice; TIMEOUT is a constant, that is the total duration of the game; expiTime is the expiration time.

bool Two booleans, gameClose if to indicate if the game is closed, if the dice are rolled gameClose will be true, so players could withdraw their reward. noPlayer indicates if there are two players in the game, if not the first player is possible to withdraw the joining fee.

event Four events are used to record the activities. Winner record the winner's address and dice number. Reveal the record player's address and key used to reveal. Commit record player's address and commit content. Withdraw record player's address and amount of withdrawal.

2 A detailed gas evaluation of your implementation, including (but not limited to):

2.1 The cost of deploying and interacting with your contract.

The gas used to deploy the contract is 2499074 units of gas.

The first player's (player1) address is 0xBE1A323bD614B7feca731aB1e3e8f9A14BE7612d. The second player's (player2) address is 0x6537e8392bF0Ce6167331FE5e46b9c2a86aFd8f2.

Activities	**Player**	**gas cost (unit of gas)**
Join game	Player1	113046
Join game	Player2	99746
Game reveal	Player1	70527
Game reveal	Player2	70527
Play game	Player1	123236
Withdraw 4 ether	Player1	109902
Withdraw 2 ether	Player2	73726

The total cost for player1 excepting the activity play game is 293475, and for player 2 is 243999. However the total cost for player1 including the activity play game is 416711.

2.2 Whether your contract is fair to both players, including whether one player has to pay more gas than the other and why.

The joining and reveal activity could be considered a gas fair. For example in section 2.1 the gas cost of the two players is close. The reveal gas fee is the same, the joining game gas cost is determined by the commit, and that is determined by the player.

However, the activity play game will cost a lot. of gas and is considered as not fair. The playGame() function can be run by both players but is only required to be run by one player, and that player will pay for the cost. The function only runs once, the boolean flag gameClose will be set to true after running.

The withdrawal gas fee could also be considered fair. It is determined by the ether, the winner withdraws more ether so more gas cost and vice versa. So the unfair gas cost is acceptable, more play gets more gas than they pay. withdrawal gas fee is determined by the ether, the winner withdraws more ether so more gas cost and vice versa.

That is the situation in which both players play the game within the time and no one receives a punishment. When one of the players does not reveal, the other player who stays in the contract will pay more gas fee for calling the function to revealGame(). But this only considers the gas fee, as the player who receives the punishment won't receive any ether back and the player who follows the rule will receive double ether.

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

Cost efficient To improve cost efficiency, the most important thing is to reduce executions. We can reduce the use of the blockchain by optimising our code, which will save us a fee on unnecessary gas costs. In the contract, there are only 4 executions that need to pay the gas fee which is considered to be necessary. The content of the function could be minimised to reduce fees, fewer variables and modifiers could be used, and easier calculations could be implemented. At the same time, the game's completeness and security also need to be ensured, which would lead to cost efficiency.

Cost fair To make the cost fairer, the contract could be designed to let the player lose the game and pay less. For example, if the player loses the game, when he/she withdraws the ether, the gas is no need to pay by him/her. Instead, the winner will pay the gas fee for the player who loses the game. The largest unfair in the contract is the playGame() function. This could also let the winner pay the gas fee calling it or let the two players divide equally. The gas fee will be added to the reward.

3 A thorough list of potential hazards and vulnerabilities that may occur in the contract. Provide a detailed analysis of the security mechanisms you use to mitigate such hazards.

Reentrancy attack Reentrancy attack occurs because the game doesn't know whether it will send the ether to an address or a contract. An attacker can easily exploit this vulnerability because all he/she

needs to do is to get some amount of the balance mapped to his/her smart contract address and create a fallback function that calls withdrawFees().

To prevent a reentrancy attack, all the other internal work should be finished before calling the withdraw function. The 'pull over push' method is used for withdrawal. transfer() and send() is better than call(). transfer() would stop the transaction and raise an exception when some errors happen, but send() and call() return a false value, not stopping the transaction. The smart contract here uses transfer(). The contract will let player access the withdrawFees() function instead of automatically sending ether to their address. Checks-Effects-Interactions pattern [2] is used in the withdrawFese(), checks the arguments, updates the state and then interacts. More specifically, set the balance of the player to zero to prevent external calls, then let the player withdraw using transfer().

Front-running Front-running is simply overtaking an unconfirmed transaction as a result of the transparency aspect of the blockchain. One of the most widespread weaknesses of the smart contract is guarded against is this one. All unconfirmed transactions are visible before they are included in a block. A few attackers can simply follow up on these transactions and overcome them by paying transaction costs.

It is very difficult to against front-running in the smart contract. The contract could be made more secure by using commit-reveal. However, the attacker might keep track of the values that are committed. The commit of the user can be front-run by an attacker, who can then post the commitment as their own. A new transaction that posts a new commitment forces the user to pay more for gas. The attacker can keep running in front of the users until they run out of money to pay for gas.

Randomness When we need to generate a random number, there are a lot of ways like using a number, block.difficult or block.hash [5]. They are all good methods, but they can be manipulated by a malicious miner, so in this contract, we want a random number but it might be controlled or edited by someone. Also, they are shared within the same block to all users because all data posted on the chain are visible. They are all on-chain data which is public. Commit-reveal scheme is the best way to solve this problem. In this contract, each player commits a hash of the random value and reveals their values. XORs the two values to produce a seed. If players are honest, then the seed is random. This could use to produce an unpredictable random number and avoid using public on-chain data, but front-running may happen.

Missing precondition checks The missing precondition checks could easily be exploited by attackers, they are the result of a sloppy design process. Although the game could run without precondition checks, it has potential risks and is difficult to be seen the designer. So when I design the game, I think of the precondition that happens in many scenarios. A lot of modifiers are used in the contract to make sure the game runs as expected. A player who is already in the game is not allowed to re-enter, there is no third player in the game and each player must pay exactly 3 ether to join. Only when there are two players and the game does not timeout they could reveal the game. The game is not allowed players to play the game unless there are two players in the game, and both players revealed the game, that ensures the players are honest and the game is not timeout. Only players could withdraw from the contract. The precondition checks also let the player play the game in the right sequence. If not join the game no reveal, no reveal then not allowed to play the game.

Integer overflow/underflow Integers are possible to overflow or underflow in the smart contract. Like the mileage counter on a car, when the system reaches the maximum value, the counter immediately resets to the initial value. The outcome is an extremely big number when subtracting two unsigned numbers, such as 4 from 3, which will also result in an underflow [4]. A safe math library, such as the SafeMath library from OpenZeppelin, could be used to avoid this. Arithmetic operations using big integers are also possible for overflow and underflow, and it could lead to a lack of precision. So instead of using 1e18 to calculate ether rewarded, I use 1 ether to reduce the risk of overflow and underflow.

4 A detailed description of the tradeoffs and choices you made, e.g., between security and performance, fairness and efficiency, etc.

Random number Except for the commit-reveal scheme mentioned in section 3 Randomness, the seed could also be combined with the hash of a future block to produce a safer seed. However, there is the probability that a malicious miner will create a future block. Considering the trade-off between workload, efficiency and gas gee, the commit-reveal scheme is safe enough to produce the seed for the random number.

Complex to play the game As mentioned above, the way to generate a random number is very complex and the game is complicated to play. The player needs to submit the commit, submit the key and salt to reveal the commit to verify the identity. To get the commit, the user needs to hash the key and salt by themselves. Although the game is secure and fair but very complex.

Punishment If one player is honest but the other is not, what should we do? Considering fairness, punishment is allowed in the game. If one player failed to reveal or did not reveal until timeout, the other player could withdraw all 6 ether. If only one player is in the game, to be fair he/she could withdraw the joining fee back. This design makes sure the player could at least get a refund, it is a fair design but not efficient for the game. The timeout is to make sure the game is not stuck. It ensures the game performs well but raises the game's complexity.

Gas cost In the contract design making the gas cost-efficient and fair is necessary. Gas is considered to be efficient, but because of the playGame() function, the gas cost is not considered to be fair. It could be fairer to let both players call the function, but in this way, the gas is not that efficient because the unnecessary function is increased. And the current design is less complex to play with.

5 Your analysis of your fellow student's contract (along with relative code snippets of their contract, where needed for readability), including (but not limited to):

The code see Section 8.

5.1 Any vulnerabilities discovered?

The contract is easier when playing. There is no commit-reveal in the contract, the number is random from the block timestamp, difficult and caller's address. Once the second player joins the game, it automatically starts and rolls the dice. After the game finished all players could withdraw their reward.

The game is easy to play, and relatively gas fair. The winner will pay for the other player's gas fee. However, the game's security is a big problem.

The contract here use

```
n = uint(keccak256(abi.encodePacked(block.timestamp, block.difficulty, msg.sender))) % 6 + 1;
```

to get the random number. However, block.timestamp and block.difficulty is easy to get by the attacker. The msg.sender could be seen by anyone in the contract.

Since there is no commit-reveal scheme used in the contract, a front-running attack could be performed here.

The contract allows the user to withdraw the reward instead of sending the reward, which is good. But a reentrancy attack could happen because the balance for each player is not set to zero before the withdrawal.

There are very few precondition checks in the contract. Anyone in any situation could call the function.

The contract has a lot of arithmetic operations using large integers. There are possibilities of integer overflow and integer underflow. Also, integer arithmetic can lead to a lack of precision.

5.2 How could a player exploit these vulnerabilities to win a game?

The player could use block.timestamp, block.difficulty and msg.sender to predict the result of the random number. He/She could join the game when it is possible to win.

A reentrancy attack could be performed by the player to withdraw the ether in the other player's balance. Although this might not be seen as 'winning a game', it does have the same effect of winning games, which is having more ether.

6 The transaction history of an execution of a game on your contract.

The first player's (player1) address is 0xBE1A323bD614B7feca731aB1e3e8f9A14BE7612d. The second player's (player2) address is 0x6537e8392bF0Ce6167331FE5e46b9c2a86aFd8f2.

The contract deployment address is 0xd8021e3Bd8034b97dB9916C0c2BeE8E3046e6B4B.

Transaction history for the first player is (include contract deployment):

[**?**].

References

[1] Keccak-256 online. https://emn178.github.io/online-tools/keccak$_2$56.*html*.

[2] Security considerations — solidity 0.5.1 documentation. https://docs.soliditylang.org/en/v0.5.1/security-considerations.htmlre-entrancy.

[3] adibas03. Online ethereum abi encoder. https://adibas03.github.io/online-ethereum-abi-encoder-decoder//encode.

[4] C. Editor. 5 common vulnerabilities in smart contract | cystack security. https://cystack.net/blog/5-common-vulnerabilities-in-smart-contract, 06 2022.

[5] Y. M. A. Quddus. How to generate random numbers in solidity – finxter. https://blog.finxter.com/how-to-generate-random-numbers-in-solidity/.