

1. Introduction

Self-Test for this game using Remix VM, so that I have lots of accounts to test with my smart contract.

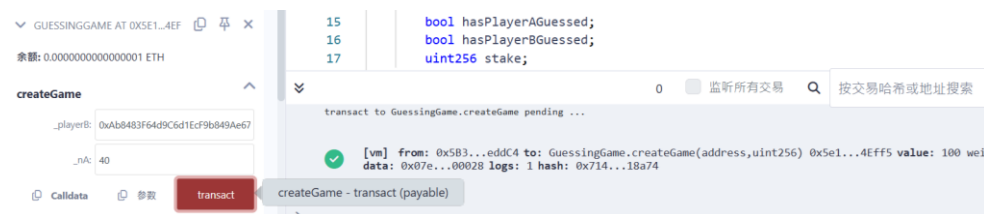
(Player A: 0x5B3...eddC4 / Player B: 0xAb8...35cb2, both starts with 100 ETH)

- a. Deploy the smart contract (2,626,462 wei to deploy this contract)



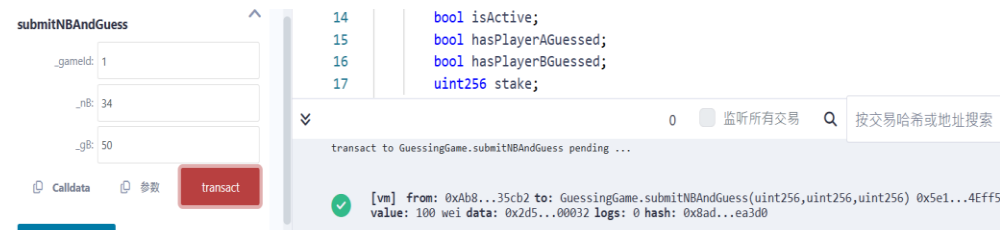
0x5B3...eddC4 (99.9999999999997373538 ether)

- b. Set 100 wei as stake, player A create this game by using **createGame()**, player A has to input nA at this stage. (330,178 wei to create this game)



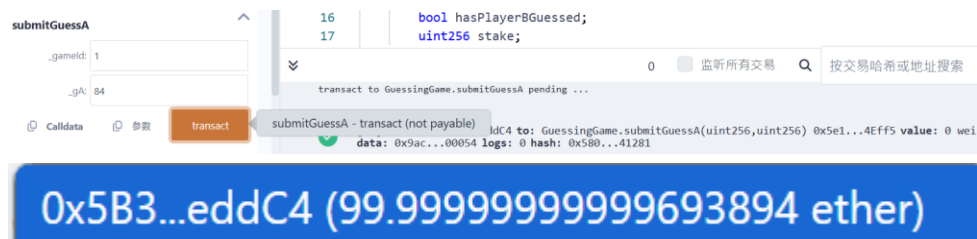
0x5B3...eddC4 (99.999999999999704336 ether)

- c. Now, player B submit nB, gB at this stage. This is the first game, so game_id is 1. He also has to pay 100 wei for stake. (157,974 wei to submit his answer)



0xAb8...35cb2 (99.999999999999842026 ether)

d. Now it turns back to player A to hand in his guessing. (104,420 wei)



e. After all guessing finished, we can resolve this game.



f. At this time, we can see the money has successfully transferred to the winner player B. ($n_A + n_B = 74,842,100 - 842,026 = 74$, matched!)

0xA8...35cb2 (99.99999999999998421 ether)

2. High-Level Design Decisions

- **Who pays for the reward?**

Both Player A and player B equally contribute to the reward pool by providing the same stake amount. Player A initiates the game with a stake, and Player B matches this stake when they join. This pooled Ether forms the reward for the winner and is held within the contract until the game is resolved.

- **How is the Reward Sent to the Winner?**

The reward (calculated as the sum of the chosen numbers $n_A + n_B$) is transferred directly to the winning player's address via the **resolveGame()** function, which uses **payable(address).transfer()** to handle payouts securely. In the event of a draw, each player's stake is refunded.

- **How is it guaranteed that a player cannot cheat?**

- **Reentrancy Guard package:** The nonReentrant modifier import from OpenZeppelin's **ReentrancyGuard** prevents reentrancy attacks, ensuring that a function cannot be called while it's still executing. (This idea was found on the website)

- **Initial Stake Matching:** Both players must provide an equal stake to join the game to ensure financial commitment. Funds are locked within the contract, preventing unauthorized withdrawal.
- **Sequential Guessing Control:** The contract prevents one player from influencing the other's guesses by enforcing sequential guessing: Player B submits their number and guess, then Player A submits their guess. Once submitted, it cannot be changed, which is enforced through the **hasPlayerAGuessed()** flag.
- **Game Completion Checks:** The game requires both players to have submitted their guesses before determining a winner, ensuring no player can back out or alter inputs post-submission.
- **What data type/structures did you use and why?**
 - **Events:** This provides transparent logging of game state changes, aiding in debugging and user notifications.
 - **struct Game:** Organizes all game-related data for each unique game ID, making data retrieval efficient.
 - **Mapping:** Maps each game ID to its respective Game struct, allowing O(1) time complexity for access.
 - **uint256:** are used for numbers and IDs to optimize gas usage and compatibility with other blockchain functions.

3. Gas Evaluation

- **Cost of Deploying and Interacting:**
 - **Deployment:** It incurs standard gas costs based on the contract size, event structures. Including the **ReentrancyGuard** increases the contract size slightly but is crucial for security. (2,626,462 wei for gas)

0x5B3...eddC4 (99.99999999997373538 ether)

- **CreateGame:** Player A covers the initial stake, ensuring commitment to the game. The gas cost for this function is relatively low as it involves storing struct data in the mapping. (330,178 wei for gas)

0x5B3...eddC4 (99.9999999999704336 ether)

- **submitNumberAndGuess/submitGuess:** Both functions are designed to limit repeated guesses, lowering the gas usage per player by handling each transaction only once. (104,420 wei for gas)

0x5B3...eddC4 (99.99999999999693894 ether)

- **Resolving Game:** This function entails moderate gas usage due to conditional checks and transfer statements but avoids complex calculations, keeping costs fair. (105,978 wei for gas)

0x5B3...eddC4 (99.999999999996832962 ether)

- **Fairness:**

Both players pay similar gas amounts for their respective function calls, with Player A paying slightly more to create the game. Subsequent interactions are designed to keep gas costs equitable, with minor variation depending on the winner transfer path.

- **Techniques to make your contract more cost efficient and/or fair:**

- **Modifier Usage (Efficient):** Checking conditions by using modifiers, avoiding repetitive code and reducing gas costs.
- **Minimizing Storage Writes (Efficient):** Events are used instead of storage variables where possible to reduce gas costs. Using events to log critical information rather than storing it in state variables, reducing storage costs.
- **ReentrancyGuard (Fair):** Ensures that interactions with external accounts are protected against reentrancy, minimizing risk and potential gas waste from re-executed transactions.

4. Potential Hazards and Security Mechanisms

- **Stake Manipulation** -> Players are required to submit matching stakes, non-compliance results in a reverted transaction. By requiring exact match of stakes, this game can start working properly.
- **Guess Tampering** -> Can only submit 1 time and verify game activity.

- **Race Conditions** -> Ensured by requiring both players to submit their guesses before resolving the game.
- **DoS through High Gas Usage** -> Each function's logic is optimized to prevent excessive gas consumption, but gas limits on the Ethereum network may restrict certain users from executing transactions in highly congested conditions.

5. Tradeoff Decisions:

- **Transparency vs. Gas Cost:** Using events adds gas costs but provides essential transparency and logging for debugging and audit trails.
- **Security vs. Performance:** Ensuring both players cannot alter game values post-submission prioritizes security over minimal gas efficiency. However, it enhances game integrity.
- **Fairness vs. Cost Efficiency:** Fairness is maintained by requiring equal stakes from both players and enforcing a draw mechanism, even though it may slightly increase complexity and gas use.

6. Comparison with a Fellow Student's Contract:

I have played the contract: 0x8Cb893af7d9C2C5b2CF8f9ff6C6aC1c7400f5e98, and I found that there's some problems compared to my code:

- **Complexity of Game State Management:**
He introduces multiple states (WaitingForPlayers, WaitingForHashes, WaitingForGuesses, WaitingForNumbers, WithdrawState), increasing complexity and potential points of failure.
My code streamlines the state management by relying on a few boolean flags, leading to a simpler and more robust implementation.
- **Gas Efficiency:**
His code involves multiple state changes and additional transactions (hash submission, number revelation), which can increase gas costs.

My code aims to minimize state changes and transactions, potentially leading to lower gas usage.

- **Refund Mechanism:**

His code utilizes a refund mapping, requiring players to manually withdraw their funds. This introduces an extra step to be attacked or cheated and potential delays in receiving funds.

My code directly transfers funds to the winners, providing a more user-friendly and efficient experience, and cannot leave the game while the game is not finished, preventing cheating.

7. Address of my Smart Contract

Address: 0x2917dc10554A1Baf244578806BA11e3184927271

<https://sepolia.etherscan.io/address/0x2917dc10554A1Baf244578806BA11e3184927271>

Here's my test transaction:

Transaction Hash	Method	Block	Age	From	To	Amount	Txn Fee
0x1846cff9658...	Resolve Game	7017984	26 hrs ago	0x6279CD3e...33c549a50	0x2917dc10...184927271	0 ETH	0.00052776
0xa16817c727...	Submit Guess A	7017979	26 hrs ago	0x6279CD3e...33c549a50	0x2917dc10...184927271	0 ETH	0.0005041
0x8725ce810fa...	Submit NB An...	7017976	26 hrs ago	0x656b4f5d...ddB44400a	0x2917dc10...184927271	100 wei	0.00079802
0xb27904fbaf2...	Create Game	7017965	26 hrs ago	0x6279CD3e...33c549a50	0x2917dc10...184927271	100 wei	0.00167831

And here's money transfer:

Parent Transaction Hash	Block	Age	From	To	Amount
0x8d4779e1c9...	7025285	39 secs ago	0x2917dc10...184927271	0x1bb36Cbe...2B03bAd30	74 wei

8. Smart Contract Code Explanation

First, import the online package “ReentrancyGuard” to enhance the security of my smart contract. Then define all parameters in my contract GuessingGame, as well as game status(create, end, draw). Use the modifier to check the player and game status, popping out error messages if necessary. Finally, by using payable function in constructor, it can complete the function of transferring money. That's all for the contract preparation.

```

1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.0;
3  // import package to make my contract more secure
4  import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
5
6  contract GuessingGame is ReentrancyGuard {
7      struct Game {
8          address playerA_Addr;
9          address playerB_Addr;
10         uint256 nA; // A's chosen number
11         uint256 nB;
12         uint256 gA; // A's guess
13         uint256 gB;
14         bool isActive; // check if the game is active
15         bool hasPlayerAGuessed; // check if A has guessed
16         bool hasPlayerBGuessed;
17         uint256 stake; // Amount staked by each player
18     }
19
20     mapping(uint256 => Game) public games; // Mapping to store games by ID
21     uint256 public gameCount; // ctr for the # of games, start from 1
22     event GameCreated(uint256 gameId, address playerA_Addr, address playerB_Addr); // circumstances: start
23     event GameEnded(uint256 gameId, address winner, uint256 reward); // circumstances: end
24     event GameDraw(uint256 gameId); // circumstances: draw
25
26     // error message: wrong player_id
27     modifier onlyPlayers(uint256 _gameId) {
28         require(
29             msg.sender == games[_gameId].playerA_Addr || msg.sender == games[_gameId].playerB_Addr,
30             "Error Message: This is Not a player in this game!"
31         );
32         _;
33     }
34
35     // error message: not start yet
36     modifier gameIsActive(uint256 _gameId) {
37         require(games[_gameId].isActive, "Error Message: This game is not active!");
38         _;
39     }
40
41     // use payable to transfer money
42     constructor() payable {}

```

First step, player A creates the game and triggers this function. It needs another player's address to ensure they are in the same game. After that, set up the status of all parameters, ensuring that the functionality will not be cheated before next step be triggered.

```

44     // Step 1. A create game with an initial stake
45     function createGame(address _playerB_Addr, uint256 _nA) external payable {
46         require(msg.value > 0, "Error Message: Player A must provide initial stake"); // error message: create game cost gas
47         gameCount++; // start from 1
48         games[gameCount] = Game({
49             playerA_Addr: msg.sender,
50             playerB_Addr: _playerB_Addr,
51             nA: _nA,
52             nB: 0,
53             gA: 0,
54             gB: 0,
55             isActive: true,
56             hasPlayerAGuessed: false,
57             hasPlayerBGuessed: false,
58             stake: msg.value
59         });
60         emit GameCreated(gameCount, msg.sender, _playerB_Addr);
61     }

```

Following up, player B accepts the game, here I need to check for more conditions (stakes, address, gB). Make sure that all information are correct, so that this contract can keep going.

```

63 // Step 2. B accept game have to pay same stake
64 function submitNBAndGuess(uint256 _gameId, uint256 _nB, uint256 _gB) external payable onlyPlayers(_gameId) gameIsActive(_
65     Game storage game = games[_gameId];
66     require(msg.sender == game.playerB_Adr, "Error Message: Wrong player! Only Player B can call this!");
67     require(msg.value == game.stake, "Error Message: You provide wrong stake! Player B must match the stake!");
68     require(!game.hasPlayerBGessed, "Error Message: Player B has already submitted your guess!");
69     game.nB = _nB;
70     game.gB = _gB;
71     game.hasPlayerBGessed = true;
72 }

```

After player B submit all his information, the controller goes back to player A, and player B will no longer to be interfere with this contract. For player A, he needs to send his guessing gA so that this contract can keep going.

```

74 // Step 3. A submit guess for B's number
75 function submitGuessA(uint256 _gameId, uint256 _gA) external onlyPlayers(_gameId) gameIsActive(_gameId) {
76     Game storage game = games[_gameId];
77     require(msg.sender == game.playerA_Adr, "Error Message: Wrong Player! Only player A can call this!");
78     require(!game.hasPlayerAGessed, "Error Message: Player A has already submitted your guess!");
79     game.gA = _gA;
80     game.hasPlayerAGessed = true;
81 }

```

Finally, after two players submitted all the data, they are no longer to take part in this contract, just waiting for the contract pending. This function is to determine the winner, as well as the winning prize that one has to receive.

```

83 // Step 4. resolve the game and reward the winner
84 function resolveGame(uint256 _gameId) external gameIsActive(_gameId) nonReentrant {
85     Game storage game = games[_gameId];
86     require(game.hasPlayerAGessed && game.hasPlayerBGessed, "Error Message: Both players must submit guesses!");
87     uint256 diffA = absDiff(game.gA, game.nB);
88     uint256 diffB = absDiff(game.gB, game.nA);
89     uint256 reward = game.nA + game.nB;
90     if (diffA < diffB) {
91         payable(game.playerA_Adr).transfer(reward);
92         emit GameEnded(_gameId, game.playerA_Adr, reward);
93     } else if (diffB < diffA) {
94         payable(game.playerB_Adr).transfer(reward);
95         emit GameEnded(_gameId, game.playerB_Adr, reward);
96     } else {
97         payable(game.playerA_Adr).transfer(game.stake);
98         payable(game.playerB_Adr).transfer(game.stake);
99         emit GameDraw(_gameId);
100     }
101     game.isActive = false;
102 }

```

Calculate absolute value, make sure that gA-nB, gB-nA always be positive.

```

104 // calculate the abs value for result
105 function absDiff(uint256 a, uint256 b) private pure returns (uint256) {
106     return a >= b ? a - b : b - a;
107 }

```

9. Run on Sepolia(Self-test)

Step.1 Use Player A account to submit the stake and playerB's address, and nA

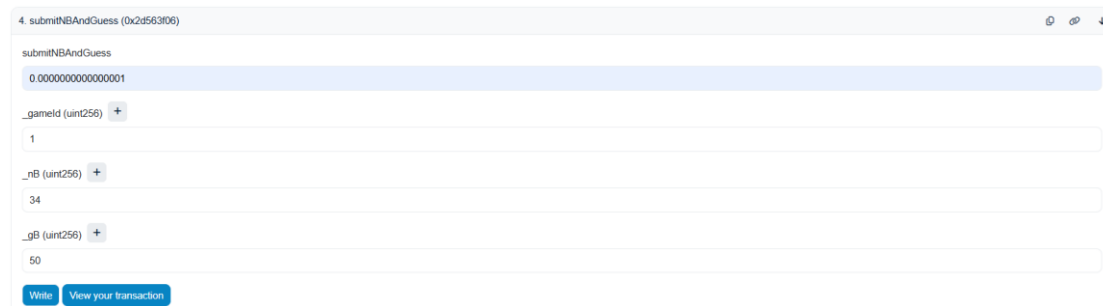
Connected - Web3 [0x1b3...ad30] [Collapse All] [Reset]

```

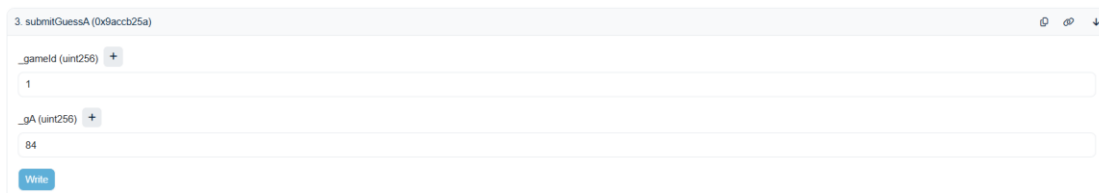
1. createGame (0x07ead930)
createGame
0.0000000000000001
_playerB (address)
0xa2742601be830bd800D8c125C77ea617ab692cdA
_nA (uint256)
40

```

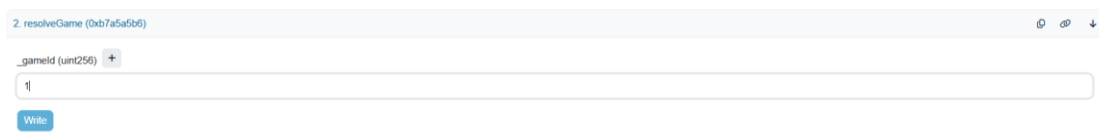

Step.2 Use Player B's account (open in a new tab) and submit the same stake and game_id (it up to the times total contract have been called), nB, gB



Step.3 Return to Player A's account to submit game_id to confirm in the same game and submit gA



Step.4 Use Player A's account to submit this game_id when finished the game



10. Reminder

1. Initial Stakes will not be refund, it acts as the participation fee.
2. If you want to test my smart contract on Sepolia, please make sure that the game_id is correct. I've played 3 times, and there's a student interacted with my contract for 1 time. Although I could not be sure that if there's anyone keep playing, but at least if you want to try my contract, start with game_id = 5, as you are playing the fourth game. Great Thanks!!
3. In my B270267.sol code, it contains online resources, but I didn't write in this report for explanation. If you want to look for original code, please visit my Sepolia test net to see the contract.