Protocol Audit Report

Azriel

April 23, 2025

Protocol Audit Report

Version 1.0

Cyfrin.io

April 23, 2025

Protocol Audit Report

Azriel

April 23, 2025

Prepared by: Azriel

Table of Contents

Protocol Summary

Simulates Rock Paper Scissors game on Web3 and ensures fairness that players are not able to view their opponent's moves ahead of time.

Disclaimer

The Azriel(me) team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

In Scope:

```
src/
--RockPaperScissors.sol - Main game contract
--WinningToken.sol - ERC20 token awarded to winners
```

Compatibilities:

Blockchains: Ethereum Mainnet All EVM-compatible chains

Tokens: ETH (for betting) RPSW (Rock Paper Scissors Winner Token) - internal ERC20 token

Roles

Players: Users who create or join games, commit and reveal moves, and participate in matches
Admin: The protocol administrator who can update timeout parameters and withdraw accumulated fees
Contract Owner: Initially the deployer of the contract, capable of setting a new admin

Executive Summary

For this audit, I used about 6h in total spread across 2 days to conduct the audit and complete the report. I made use of tools like Slither and Aderyn to first conduct static analysis, in addition to manual analysis. Additionally, I used cloc and Solidity Metrics to aid me in the initial scoping phase and understanding of the code base better, although there are only 2 files in scope.

Issues found

Severity	Number of Issues Found
High	1
Medium	2
Low	0
Info	5

Findings

[H-1]`RockPaperScissors::joinGameWithEth` Allows Users to Join Token Games Without Paying

Description: The `RockPaperScissors::joinGameWithEth` function fails to validate that msg.value > 0. As a result, an attacker can call this function with msg.value == 0 and still successfully join a Token-based Game, provided the original creator also set the bet to 0. The check require(msg.value == game.bet) passes because both values are 0.

Impact: An attacker can exploit this by joining a token game without staking any tokens. After joining, the attacker commits a move and immediately calls `timeoutReveal`, which triggers `_cancelGame` — a function that mints 1 WinningToken to both participants. Since the attacker joined without any cost, they effectively farm free tokens. Repeating this at scale enables rapid minting of arbitrary amounts of tokens, undermining the integrity and scarcity of the token economy.

Proof of Concept: To test this, I have added the following test function into the current test suite. The following proof of concept demonstrates the exact exploit highlighted above.

```
address public playerC = makeAddr("playerC");
address public playerD = makeAddr("playerD");
uint256 testGameId;

function testJoinGameWithTokenUsingEth() public {
    // Set up 2 new players, we will be using playerD's account as the attacker
    vm.prank(address(game));
```

```
        token.mint(playerC, 10);

        vm.prank(address(game));
        token.mint(playerD, 10);
        vm.stopPrank();

        // Player C first creates a token game
        vm.startPrank(playerC);
        token.approve(address(game), 1);
        testGameId = game.createGameWithToken(TOTAL_TURNS, TIMEOUT);
        vm.stopPrank();

        // Attacker joins the same game using joinGameWithEth (with 0 msg.value)
        vm.startPrank(playerD);
        game.joinGameWithEth(testGameId);
        vm.stopPrank();

        assertEq(token.balanceOf(playerC), 9);
        // verify that player D did not transfer tokens
        assertEq(token.balanceOf(playerD), 10);

        // Verify game state, ensure that player D is in the game
        (address storedPlayerC, address storedPlayerD,,,,,,,,,,,,,,) =
            game.games(testGameId);

        assertEq(storedPlayerC, playerC);
        assertEq(storedPlayerD, playerD);
        assertEq(uint256(state), uint256(RockPaperScissors.GameState.Created));

        // Commit a move for player D
        bytes32 saltD = keccak256(abi.encodePacked("salt for player D"));
        bytes32 commitD = keccak256(abi.encodePacked(uint8(RockPaperScissors.Move.Rock), sal

        vm.startPrank(playerD);
        game.commitMove(testGameId, commitD);
        // Call timeoutReveal immediately after commiting to get a the game cancelled
        game.timeoutReveal(testGameId);
        vm.stopPrank();

        // Verify balances, showing that playerD was able to gain tokens
        assertEq(token.balanceOf(playerC), 10);
        assertEq(token.balanceOf(playerD), 11);
    }
```

Recommended Mitigation: An additional check that game.bet > 0 should be included within the function. This would ensure that the function cannot be

used to join token games.

```
function joinGameWithEth(uint256 _gameId) external payable {
    Game storage game = games[_gameId];

    require(game.state == GameState.Created, "Game not open to join");
    require(game.playerA != msg.sender, "Cannot join your own game");
    require(block.timestamp <= game.joinDeadline, "Join deadline passed");
    require(msg.value == game.bet, "Bet amount must match creator's bet");
    require(game.bet > 0, "Eth must be sent to join an Eth game");

    game.playerB = msg.sender;
    emit PlayerJoined(_gameId, msg.sender);
}
```

[M-1] Funds / Tokens Could be locked up during games with unresponsive opponents, if opponent does not call `RockPaperScissors::commitMove`.

Description: When a player join a game and commits a move, the game.revealDeadline variable is not (default value 0) only until his opponent also commits a move, while the game state has already changed to `Committed`. If the opponent is unresponsive, it could cause the funds / tokens to be locked within the contract without a way to cancel.

The following reasons are why the current mechanisms for handling timeouts / cancelling games cannot handle this situation:

1. `RockPaperScissors::timeoutReveal` cannot be called as the check `require(block.timestamp > game.revealDeadline, "Reveal phase not timed out yet");` will not pass since `game.revealDeadline` is the 0 value when not set.
2. `RockPaperScissors::cancelGame` cannot be called as the game is neither in `Created` state nor is the player the game creator in the following checks: `require(game.state == GameState.Created, "Game must be in created state");` and `require(msg.sender == game.playerA, "Only creator can cancel");`

Impact: This could cause players to lose their funds / tokens to the contract without a way to rescue them.

Proof of Concept: NA

Recommended Mitigation: There could be an additional field set within games to update when one of the player has committed a move, and either players could have the option to cancel the game if their opponent has not responded within a certain time limit. The following are examples of how the code could be updated to allow for this:

```
// Game structure
```

```
    struct Game {
        address playerA; // Creator of the game
        address playerB; // Second player to join
        uint256 bet; // Amount of ETH bet
        uint256 timeoutInterval; // Time allowed for reveal
++        uint256 commitDeadline; // Deadline for committing a move
        uint256 revealDeadline; // Deadline for revealing moves
        uint256 creationTime; // When the game was created
        uint256 joinDeadline; // Deadline for someone to join the game
        uint256 totalTurns; // Total number of turns in the game
        uint256 currentTurn; // Current turn number
        bytes32 commitA; // Hashed move from player A
        bytes32 commitB; // Hashed move from player B
        Move moveA; // Revealed move from player A
        Move moveB; // Revealed move from player B
        uint8 scoreA; // Score for player A
        uint8 scoreB; // Score for player B
        GameState state; // Current state of the game
    }
```

```
    // commitMove function
    if (game.commitA != bytes32(0) && game.commitB != bytes32(0)) {
++        game.commitDeadline = 0; // Reset the deadline once both parties have committed
        game.revealDeadline = block.timestamp + game.timeoutInterval;
    }
++    else {
++        game.commitDeadline = block.timestamp + 1 days; // or whatever duration that the c
++    }
```

```
    // additional function for either player to cancel the game after one side has committe
    function timeoutCommit(uint256 _gameId) external {
        Game storage game = games[_gameId];

        require(game.state == GameState.Committed, "Game must be in created state");
        require(msg.sender == game.playerA || msg.sender == game.playerB, "Only players can
        require(game.commitDeadline != 0, "Commit deadline must be set!");
        require(block.timestamp > game.commitDeadline, "Commit deadline not reached yet");

        _cancelGame(_gameId);
    }
```

[M-2] Business logic errors in `RockPaperScissors::timeoutReveal` due to Lack of validation that `game.revealDeadline` is set, potentially causing disruptions to games

Description: Currently, `RockPaperScissors::timeoutReveal` is supposed to be used once both players have committed a move, and one or both party has

not revealed their move until the deadline, allowing the game to be forfeit or cancelled. However, due to a lack of validation of `game.revealDeadline` within the check `require(block.timestamp > game.revealDeadline, "Reveal phase not timed out yet");`, if the `game.revealDeadline` field is not set (which default value is 0) this check would always pass. This would allow players to immediately call this function once they join a game and commit a move.

Impact: An attacker could cause disruptions to current games by joining all the games, committing a move and then immediately calling `RockPaperScissors::timeoutReveal` to get the games cancelled. No funds would be lost in the process, but there could be potential business disruptions to the games.

Proof of Concept: Please refer to H-1, where this exploit was also demonstrated

Recommended Mitigation: Adding an additional check to ensure the `game.revealDeadline` would solve this issue.

```
    function timeoutReveal(uint256 _gameId) external {
        Game storage game = games[_gameId];

        require(msg.sender == game.playerA || msg.sender == game.playerB, "Not a player in t
        require(game.state == GameState.Committed, "Game not in reveal phase");
++        require(game.revealDeadline != 0, "Reveal Deadline not set yet");
        require(block.timestamp > game.revealDeadline, "Reveal phase not timed out yet");

        // If player calling timeout has revealed but opponent hasn't, they win
        bool playerARevealed = game.moveA != Move.None;
        bool playerBRevealed = game.moveB != Move.None;

        if (msg.sender == game.playerA && playerARevealed && !playerBRevealed) {
            // Player A wins by timeout
            _finishGame(_gameId, game.playerA);
        } else if (msg.sender == game.playerB && playerBRevealed && !playerARevealed) {
            // Player B wins by timeout
            _finishGame(_gameId, game.playerB);
        } else if (!playerARevealed && !playerBRevealed) {
            // Neither player revealed, cancel the game and refund
            _cancelGame(_gameId);
        } else {
            revert("Invalid timeout claim");
        }
    }

    function canTimeoutReveal(uint256 _gameId) external view returns (bool canTimeout, addre
        Game storage game = games[_gameId];

--        if (game.state != GameState.Committed || block.timestamp <= game.revealDeadline) {
```

```
++          if (game.state != GameState.Committed || (game.revealDeadline != 0 && block.timest
              return (false, address(0));
          }

          bool playerARevealed = game.moveA != Move.None;
          bool playerBRevealed = game.moveB != Move.None;

          if (playerARevealed && !playerBRevealed) {
              return (true, game.playerA);
          } else if (!playerARevealed && playerBRevealed) {
              return (true, game.playerB);
          } else if (!playerARevealed && !playerBRevealed) {
              return (true, address(0)); // Both forfeit
          }

          return (false, address(0));
      }
```

[I-1]: Unsafe ERC20 Operations should not be used

ERC20 functions may not behave as expected. For example: return values are not always meaningful. It is recommended to use OpenZeppelin's SafeERC20 library.

2 Found Instances

- Found in src/RockPaperScissors.sol Line: 131

  ```
  winningToken.transferFrom(msg.sender, address(this), 1);
  ```

- Found in src/RockPaperScissors.sol Line: 180

  ```
  winningToken.transferFrom(msg.sender, address(this), 1);
  ```

[I-2]: Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

2 Found Instances

- Found in src/RockPaperScissors.sol Line: 2

  ```
  pragma solidity ^0.8.13;
  ```

- Found in src/WinningToken.sol Line: 2

  ```
  pragma solidity ^0.8.13;
  ```

[G-1]: `public` functions not used internally could be marked `external`

Instead of marking a function as `public`, consider marking it as `external` if it is not used internally.

2 Found Instances

- Found in src/RockPaperScissors.sol Line: 378

```
function tokenOwner() public view returns (address) {
```

- Found in src/WinningToken.sol Line: 25

```
function decimals() public view virtual override returns (uint8) {
```

[G-2]: Define and use `constant` variables instead of using literals

If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract.

4 Found Instances

- Found in src/RockPaperScissors.sol Line: 100

```
require(_timeoutInterval >= 5 minutes, "Timeout must be at least 5 minutes");
```

- Found in src/RockPaperScissors.sol Line: 128

```
require(_timeoutInterval >= 5 minutes, "Timeout must be at least 5 minutes");
```

- Found in src/RockPaperScissors.sol Line: 483

```
uint256 fee = (totalPot * PROTOCOL_FEE_PERCENT) / 100;
```

- Found in src/RockPaperScissors.sol Line: 520

```
uint256 fee = (totalPot * PROTOCOL_FEE_PERCENT) / 100;
```

[G3]: Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

8 Found Instances

- Found in src/RockPaperScissors.sol Line: 72

```
event GameCreated(uint256 indexed gameId, address indexed creator, uint256 bet, uin
```

- Found in src/RockPaperScissors.sol Line: 74

```
event MoveCommitted(uint256 indexed gameId, address indexed player, uint256 current
```

- Found in src/RockPaperScissors.sol Line: 75

```
event MoveRevealed(uint256 indexed gameId, address indexed player, Move move, uint2
```

- Found in src/RockPaperScissors.sol Line: 76

```
event TurnCompleted(uint256 indexed gameId, address winner, uint256 currentTurn);
```

- Found in src/RockPaperScissors.sol Line: 77

```
event GameFinished(uint256 indexed gameId, address winner, uint256 prize);
```

- Found in src/RockPaperScissors.sol Line: 79

```
event JoinTimeoutUpdated(uint256 oldTimeout, uint256 newTimeout);
```

- Found in src/RockPaperScissors.sol Line: 80

```
event FeeCollected(uint256 gameId, uint256 feeAmount);
```

- Found in src/RockPaperScissors.sol Line: 81

```
event FeeWithdrawn(address indexed admin, uint256 amount);
```