TheLastManStanding Audit Report

Version 1.0

Anonymous961

August 17, 2025

The Last Man Standing Audit Report

Anonymous961

Aug 15, 2025

Prepared by: Anonymous961 Lead Auditors:

- anonymous961

Table of Contents

    - [H-3] Grace Period Bypass Allows Throne Theft (Game Logic Failure)
- Mediums
    - [M-1] New Player Can Never Enter Game Due to Inverted Claim Logic
    - [M-2] Previous King Payout Not Implemented (Protocol Deviation)
- Lows
    - [L-1] Incorrect Prize Emission in `declareWinner()` (Data Corruption)

Protocol Summary

The Last Man Standing Game is a decentralized "King of the Hill" style game implemented as a Solidity smart contract on the Ethereum Virtual Machine (EVM). It creates a competitive environment where players vie for the title of "King" by paying an increasing fee. The game's core mechanic revolves around a grace period: if no new player claims the throne before this period expires, the current King wins the entire accumulated prize pot.

Disclaimer

The anonymous961 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

The findings described in this document correspond the following commit hash:

```
de4551a88a190a21cf2a869dcf38a850757654a7
```

Scope

```
./src/
-- Game.sol
```

Roles

This protocol includes the following roles:

1. Owner (Deployer)

Powers: * Deploys the `Game` contract. * Can update game parameters: `gracePeriod`, `initialClaimFee`, `feeIncreasePercentage`, `platformFeePercentage`. * Can `resetGame()` to start a new round after a winner has been declared. * Can `withdrawPlatformFees()` accumulated from claims. **Limitations:** * Cannot claim the throne if they are already the current king. * Cannot declare a winner before the grace period expires. * Cannot reset the game if a round is still active.

2. King (Current King)

Powers: * The last player to successfully `claimThrone()`. * Receives a small payout from the next player's `claimFee` (if applicable). * Wins the entire `pot` if no one claims the throne before the `gracePeriod` expires. * Can `withdrawWinnings()` once declared a winner. **Limitations:** * Must pay the current `claimFee` to become king. * Cannot claim the throne if they are already the current king. * Their reign is temporary and can be overthrown by any other player.

3. Players (Claimants)

Powers: * Can `claimThrone()` by sending the required `claimFee`. * Can become the `currentKing`. * Can potentially win the `pot` if they are the last king when the grace period expires.

Limitations: * Must send sufficient ETH to match or exceed the `claimFee`. * Cannot claim if the game has ended. * Cannot claim if they are already the current king.

4. Anyone (Declarer)

Powers: * Can call `declareWinner()` once the `gracePeriod` has expired. **Limitations:** * Cannot declare a winner if the grace period has not expired. * Cannot declare a winner if no one has ever claimed the throne. * Cannot declare a winner if the game has already ended.

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	0
Gas	0
Total	6

Findings

High

[H-1] Excess ETH Payments Not Refunded (Funds Loss)

Description

- Users can send more ETH than `claimFee` when claiming the throne

- The contract keeps the full amount without refunding the excess

- This creates unfair overpayments and distorts game economics

```
function claimThrone() external payable {
    @> require(msg.value >= claimFee, "Insufficient ETH"); // No upper
        bound check
    // ...
    @> amountToPot = sentAmount - currentPlatformFee; // Uses full msg.
        value
    @> pot = pot + amountToPot; // Excess ETH goes to pot
}
```

Risk

Likelihood:
* High - Any user can accidentally or intentionally overpay
* Especially problematic with frontends that don't validate amounts

Impact:
* Permanent loss of excess ETH for users
* Unfair advantage for players who overpay (larger pot contributions)
* Distorts game incentives and prize calculations

Proof of Concept

Here I'm checking for 2 things - if the `claimThrone` funciton is taking more than requried `claimFee` or not. - Then if the function is taking more than the required fee then it is returning the excess funds or not.

Code

```
    function test_excessClaimFee() public {
        uint256 initialClaimFee = game.claimFee();
        console.log("claimFee is ", initialClaimFee);

        address user = makeAddr("user");

        vm.deal(user, 10 ether);
        uint256 userInitialBalance = user.balance;

        vm.prank(user);
        // 5 ether is the excess fee
        game.claimThrone{value: initialClaimFee + 5 ether}();

        // checking for current king
        assert(game.currentKing() == user);

        // excess fee is not refunded
        assert(user.balance < userInitialBalance - initialClaimFee);
    }
```

Recommended Mitigation

```
function claimThrone() external payable gameNotEnded nonReentrant {
    require(msg.value >= claimFee, "Insufficient ETH");
+   if (msg.value > claimFee) {
+       payable(msg.sender).transfer(msg.value - claimFee);
+   }

    uint256 sentAmount = claimFee; // Use exact fee amount
    // ... rest of logic ...
}
```

Here's the vulnerability report for the unfair claim fee increase mechanism:

[H-2] Claim Fee Increase Penalizes Overpayers (Game Fairness)

Description

- The fee increase calculation uses the base `claimFee` rather than the actual paid amount
- Players who pay more than minimum get no proportional benefit
- Creates perverse incentives against strategic overpayment

```
function claimThrone() external payable {
    @> claimFee = claimFee + (claimFee * feeIncreasePercentage) / 100;
       // Ignores msg.value
    // ...
    emit ThroneClaimed(msg.sender, sentAmount, claimFee, pot, block.
       timestamp);
}
```

Risk

Likelihood:
* Certain - Affects every throne claim
* Especially impacts high-value players

Impact:
* Disincentivizes players from contributing more than minimum
* Reduces potential pot growth
* Unfair to players who support the game economy

Proof of Concept

Test

```
    function test_userWithHigherDepositThrownByLowerDeposit() public {
        vm.prank(player1);
        // initial claim fee is 0.1 ether
        game.claimThrone{value: 1 ether}();

        vm.prank(player2);
        // 10% increase in 0.1 ether claim fee
        // to be exact this throne can be claimed with just 0.11 ether
        game.claimThrone{value: 0.2 ether}();

        assert(game.currentKing() == player2);
    }
```

Recommended Mitigation

```
function claimThrone() external payable gameNotEnded nonReentrant {
    require(msg.value >= claimFee);

    // Calculate fee increase based on paid amount
-   claimFee = claimFee + ((claimFee * feeIncreasePercentage) / 100);
+   claimFee = amountToPot + ((amountToPot * feeIncreasePercentage) /
   100);

    // ... rest of logic ...
}
```

[H-3] Grace Period Bypass Allows Throne Theft (Game Logic Failure)

Description

- The `claimThrone()` function lacks validation of the grace period expiration
- Allows new claims even after the current king's grace period has ended
- Violates core game mechanic that should crown the current king as winner

```
function claimThrone() external payable gameNotEnded nonReentrant {
    @> // Missing: require(block.timestamp <= lastClaimTime +
        gracePeriod)
    require(msg.value >= claimFee);
    require(msg.sender != currentKing);
    // ...
}
```

Risk

Likelihood:
* High - Any player can monitor and exploit expired grace periods
* Especially damaging in low-activity game periods

Impact:
* Steals rightful winnings from legitimate kings
* Destroys game fairness and trust
* Enables last-second throne sniping

Proof of Concept

```
    function test_userEntersTheGameAfterGracePeriodisOver() public {
        vm.prank(player1);
        // player 1 claims the throne
        game.claimThrone{value: INITIAL_CLAIM_FEE}();

        uint256 newClaimFee = game.claimFee();

        // throne claim time ends
        vm.warp(game.lastClaimTime() + GRACE_PERIOD + 1 hours);

        vm.prank(player2);
        // now player 2 claims the throne
        game.claimThrone{value: newClaimFee}();

        assert(game.currentKing() == player2);
    }
```

Recommended Mitigation

```
function claimThrone() external payable gameNotEnded nonReentrant {
    require(msg.value >= claimFee);
    require(msg.sender != currentKing);
+   require(
+       block.timestamp <= lastClaimTime + gracePeriod,
+       "Grace period expired - declare winner"
+   );
    // ...
}
```

Mediums

[M-1] New Player Can Never Enter Game Due to Inverted Claim Logic

Description

The `claimThrone()` function's access control check is inverted:

```
require(msg.sender == currentKing, "Game: You are already the king. No
    need to re-claim.");
```

This causes: 1. First claims to revert (when `currentKing == address(0)`) 2. Incorrect error message about reclaims 3. Complete lockout of new players

Impact

- **Game Never Starts**: No one can become the first king - **ETH Locked**: Funds sent to claim throne become stuck - **Core Mechanism Broken**: "Last man standing" gameplay impossible

Proof of Concept

Test

```
function testFirstClaimReverts() public {
    // new player
    address player = makeAddr("player");
    // feeding some ether to participate
    vm.deal(player, 5 ether);
    // getting initial claimFee to enter in game
    uint256 claimFee = game.claimFee();

    console.log("current king before player enters :", game.currentKing
        ());
    vm.prank(player);
    vm.expectRevert("Game: You are already the king. No need to re-
        claim.");
    // trying to claim the throne
    game.claimThrone{value: claimFee}();
    console.log("current king:", game.currentKing());
}
```

Recommended Mitigation

```
require(msg.sender != currentKing, "Game: You are already the king. No
    need to re-claim.");
```

[M-2] Previous King Payout Not Implemented (Protocol Deviation)

Description

The `claimThrone()` function fails to distribute rewards to the previous king as specified in the NatSpec comments:

```
/**
```

```
 * @dev If there's a previous king, a small portion of the new claim fee
     is sent to them.
 */
```

Key issues: 1. `previousKingPayout` is hardcoded to `0` 2. No funds are transferred to the previous king 3. Pot calculation doesn't account for king's share

Risk

Likelihood: * Occurs on every throne claim after the first one * Permanently affects all game rounds where multiple claims occur

Impact: * Violates promised game mechanics and economic model * Disincentivizes players from becoming king * Unfairly advantages subsequent claimants * Reduces protocol's trustworthiness

Proof of Concept

1. **Test Setup**:

   - Two players with 10 ETH each
   - First claim makes king1 the king (0.1 ETH fee)
   - Second claim should pay king1

2. **Key Verification**:

   ```
   assertEq(king1.balance, king1BalanceBefore); // No payout received
   assertEq(pot, potBefore + (claimFee - platformFee)); // Full
       amount goes to pot
   ```

3. **What Test Shows**:

   - Platform fee works (5%)
   - Missing previous king's payout
   - Pot gets 95% instead of receiving after the cut of previous king

Code

```
function testMissingPreviousKingPayout() public {
    address king1 = makeAddr("king1");
    address king2 = makeAddr("king2");
    vm.deal(king1, 10 ether);
    vm.deal(king2, 10 ether);

    // First claim - king1 becomes king
    uint256 firstClaimFee = game.claimFee();
    vm.prank(king1);
    game.claimThrone{value: firstClaimFee}();

    // Get balances before second claim
```

```
    uint256 king1BalanceBefore = king1.balance;
    uint256 potBefore = game.pot();
    uint256 platformFeesBefore = game.platformFeesBalance();

    // Second claim - king2 claims throne
    uint256 secondClaimFee = game.claimFee();
    vm.prank(king2);
    game.claimThrone{value: secondClaimFee}();

    // Verify NO payout to previous king (current behavior)
    assertEq(
        king1.balance, king1BalanceBefore, "Previous king should not receive payout (as currently implemented)"
    );

    // Verify platform fee was taken (5%)
    uint256 expectedPlatformFee = (secondClaimFee * game.platformFeePercentage()) / 100;
    assertEq(
        game.platformFeesBalance(), platformFeesBefore + expectedPlatformFee, "Platform fee not properly collected"
    );

    // Verify pot received remaining 95%
    assertEq(
        game.pot(),
        potBefore + (secondClaimFee - expectedPlatformFee),
        "Pot should receive claim amount minus platform fee"
    );
}
```

Recommended Mitigation

```
function claimThrone() external payable gameNotEnded nonReentrant {
    require(msg.value >= claimFee);
    require(msg.sender != currentKing);

    uint256 sentAmount = msg.value;
+   uint256 previousKingPayout = 0;
+
+   if (currentKing != address(0)) {
+       previousKingPayout = (sentAmount * 10) / 100; // Example: 10% to previous king
+       payable(currentKing).transfer(previousKingPayout);
+   }

    uint256 currentPlatformFee = (sentAmount * platformFeePercentage) / 100;

    // Update calculations to account for king's share
-   uint256 amountToPot = sentAmount - currentPlatformFee;
+   uint256 amountToPot = sentAmount - currentPlatformFee -
    previousKingPayout;

    platformFeesBalance += currentPlatformFee;
    pot += amountToPot;

    // State updates...
}
```

Lows

[L-1] Incorrect Prize Emission in `declareWinner()` (Data Corruption)

Description

The `declareWinner()` function emits incorrect prize amount in `GameEnded` event:

```
pendingWinnings[currentKing] += pot;  // pot value captured here
pot = 0;                             // pot reset to 0
emit GameEnded(currentKing, pot, ...); // Emits 0 instead of actual
    prize
```

Risk

Impact: - Frontends/analytics will display wrong prize amount (0) - Discrepancy between actual winnings and logged event - Breaks prize transparency

Likelihood: - Affects every winner declaration - Persistent data corruption in event logs

Proof of Concept

Key Test Behaviors: 1. **Initial Setup**
- Creates a player who becomes king with a known pot amount
- Captures the actual pot value (e.g., 0.95 ETH from a 1 ETH claim after 5% fee)

2. **Time Manipulation**

   - Fast-forwards past grace period using `vm.warp()`

   - Ensures winner declaration is possible

3. **Event Verification**

   - Explicitly expects emission of `0` as prize amount

- Will fail if contract emits correct pot value
- Uses Foundry's `expectEmit` for precise event matching

```
    function testPrizeEmissionIncorrect() public {
        address player = makeAddr("player");
        vm.deal(player, 1 ether);
        uint256 initialClaimFee = game.claimFee();

        // Setup game with existing king and pot
        vm.prank(player);
        game.claimThrone{value: initialClaimFee}();
        uint256 actualPot = game.pot(); // Store actual pot value

        // Fast-forward past grace period
        vm.warp(block.timestamp + game.gracePeriod() + 1);

        uint256 gameRound = game.gameRound();

        // Declare winner
        vm.expectEmit(true, true, true, true);
        emit GameEnded(player, 0, block.timestamp, gameRound); //
           Should emit actual pot
        game.declareWinner();

        assertNotEq(actualPot, 0, "Pot value is wrong");
    }
```

Recommended Mitigation

```
function declareWinner() external gameNotEnded {
    require(currentKing != address(0));
    require(block.timestamp > lastClaimTime + gracePeriod);

    gameEnded = true;
+   uint256 prize = pot;
    pendingWinnings[currentKing] += pot;
    pot = 0;

-   emit GameEnded(currentKing, pot, block.timestamp, gameRound);
+   emit GameEnded(currentKing, prize, block.timestamp, gameRound);
}
```

Fix Benefits: 1. Preserves actual prize amount in logs 2. Maintains event emission after state changes 3. Single-line change with no side effects