Last Man Standing Game Audit Report

Version 1.0

Tanu Gupta

August 4, 2025

Game Protocol Audit Report

Tanu Gupta

August 4, 2025

Prepared by: Tanu Gupta

Lead Security Researcher:

- Tanu Gupta

Table of Contents

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 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
Likelihood	High	H	H/M	M
	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 repository Last Man Standing

Scope

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

Roles

This protocol includes the following roles:

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.

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.

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.

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

Vulnerabilities have been reported using the foundry framework.

Issues found

Severity	Number of issues found
High	1
Medium	1
Low	2
Info	1
Gas	2
Total	7

Findings

High

[H-1] Critical Logic Error in `Game::claimThrone()` renders entire contract unusable and permanently locks all funds

Description The `Game::claimThrone()` function contains a fatal logic error in its access control check.

The require statement

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

should use `!=` instead of `==`. Since currentKing is initialized to `address(0)` and msg.sender can never be ‘address(0), this condition will always fail, making the function permanently uncallable.

This creates a **cascading** failure that breaks the entire contract:

- No one can ever claim the throne
- `declareWinner()` can never be called (`requires currentKing != address(0)`)

- `resetGame()` can never be called (`requires gameEnded = true` from `declareWinner()`)
- The contract becomes a fund sink with no recovery mechanism

Impact

1. Complete loss of functionality
2. Permanent fund locking
3. Contract DOA (Dead on arrival)
4. No recover possible - even the owner cannot reset or fix the contract state

Proof of Concepts

A user with balance more than `claimFee` needed to `claimThone()` is unable to claim throne.

```
function test_claim_throne() external {
        uint256 amountNeededToClainThrone = game.claimFee();
        uint256 player1Balance = player1.balance;
        assertGt(player1Balance, amountNeededToClainThrone);
        vm.startPrank(player1);
        vm.expectRevert("Game: You are already the king. No need to re-
            claim.");
        game.claimThrone{value: INITIAL_CLAIM_FEE}();
        vm.stopPrank();
    }
```

Recommended mitigation Change == to != in the require statement of access control

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

Medium

[M-1] MEV Front-Running Attack Enables Griefing and Potential Theft Through Grace Period Manipulation by superseding `declareWinner` with `claimThrone`

Note: *This vulnerability assumes the critical logic error in `claimThrone()` is fixed (changing `require(msg.sender == currentKing)` to `require(msg.sender != currentKing)`) to make the contract functional.*

Description An attacker can monitor the mempool for `declareWinner()` transactions and front-run them with `claimThrone()` calls to reset the grace period timer. This attack prevents legitimate

winners from claiming their prize and forces additional waiting periods. If no other players intervene during the new grace period, the attacker can eventually claim the entire pot themselves.

The attack exploits the fact that `claimThrone()` updates `lastClaimTime = block.timestamp`, which resets the grace period countdown and causes the original `declareWinner()` transaction to revert with **Grace period has not expired yet.**

Prerequisites

- The fundamental `claimThrone()` logic error must be fixed for the contract to function
- Game must be in progress with an active currentKing
- Grace period must be near expiration

Impact

1. Legitimate winners can not claim victory when the grace period should have expired.
2. Forcing all players to wait additional grace periods, wasting time and gas.
3. If the attacker successfully prevents others from claiming during the new grace period, they can steal the entire accumulated pot
4. Repeated attacks can theoretically extend the game indefinitely

Proof of Concepts

1. Three players played to claim thone. 3rd player being the last one becomes the potential legitimate winner.
2. After passing of the grace period, some one tries to call the `declareWinner` function.
3. However, a malicious attack sees this transaction in the mempool and supersedes this with another `claimThrone` transaction.
4. Hence manipulating the grace period and forcing players to wait for additional grace period.

```
function test_FrontRun_declareWinner_To_Cause_Grief() external {
        vm.prank(player1);
        game.claimThrone{value: INITIAL_CLAIM_FEE}();

        uint256 claimFee = game.claimFee();
        vm.prank(player2);
        game.claimThrone{value: claimFee}();

        claimFee = game.claimFee();
        vm.prank(player3);
        game.claimThrone{value: claimFee}();

        address currentKing = game.currentKing();
        assertEq(currentKing, player3);

        uint256 newTime = block.timestamp + game.getRemainingTime();
```

```
17
18          vm.warp(newTime + 1);
19
20          //declare winner
21          // game.declareWinner();
22          // uint256 winnerPendings = game.pendingWinnings(player3);
23          // assertGt(winnerPendings, 0); //3.144e17
24
25          //attacker supersedes this above transaction with claimThrone
26          claimFee = game.claimFee();
27          vm.prank(maliciousActor);
28          game.claimThrone{value: claimFee}();
29          currentKing = game.currentKing();
30          assertEq(currentKing, maliciousActor);
31
32          vm.expectRevert("Game: Grace period has not expired yet.");
33          game.declareWinner();
34          uint256 winnerPendings = game.pendingWinnings(player3);
35          assertEq(winnerPendings, 0);
36
37          //attacker is required for grace period to pass meanwhile there
                 is a chance for others to claim throne, hence causing more
                delays
38
39          newTime = block.timestamp + game.getRemainingTime();
40          vm.warp(newTime + 1);
41
42          //If no one claims the throne in between then attacker becomes
                the kind
43          game.declareWinner();
44          winnerPendings = game.pendingWinnings(maliciousActor);
45          assertGt(winnerPendings, 0); //4.408e17
46          assertEq(currentKing, maliciousActor);
47
48      }
```

Recommended mitigation Implement a commit-reveal scheme or time-lock mechanism to prevent last-second interventions. Something like this -

```
1  uint256 public claimCutoffPeriod = 1 hours; // No claims allowed in
       final hour
2
3  function claimThrone() external payable gameNotEnded nonReentrant {
4      require(
5          block.timestamp < lastClaimTime + gracePeriod -
               claimCutoffPeriod,
6          "Game: Claims disabled in final period before winner
               declaration"
7      );
8      // ... rest of function
9  }
```

Low

[L-1] No index variable defined for important event `Game::GameReset`

Description The `GameReset` event lacks **indexed** parameters, which significantly reduces its utility for off-chain applications and monitoring systems. Currently, the event is defined as:

```
    event GameReset(uint256 newRound, uint256 timestamp);
```

Impact Off-chain applications cannot efficiently filter events by specific game rounds, causing difficulty in tracking.

Recommended mitigation

```
-         event GameReset(uint256 newRound, uint256 timestamp);
+         event GameReset(uint256 indexed newRound, uint256 timestamp);
```

[L-2] CEI Pattern Violation in withdrawWinnings()

Description The function violates Checks-Effects-Interactions (CEI) pattern but is protected by non-Reentrant modifier.

```
function withdrawWinnings() external nonReentrant {
        uint256 amount = pendingWinnings[msg.sender];
        require(amount > 0, "Game: No winnings to withdraw.");

        (bool success, ) = payable(msg.sender).call{value: amount}("");
        require(success, "Game: Failed to withdraw winnings.");

        pendingWinnings[msg.sender] = 0;

        emit WinningsWithdrawn(msg.sender, amount);
    }
```

Recommended mitigation Follow CEI pattern for code clarity and defense-in-depth.

```
function withdrawWinnings() external nonReentrant {
        uint256 amount = pendingWinnings[msg.sender];
        require(amount > 0, "Game: No winnings to withdraw.");
+       pendingWinnings[msg.sender] = 0;
+       emit WinningsWithdrawn(msg.sender, amount);

        (bool success, ) = payable(msg.sender).call{value: amount}("");
        require(success, "Game: Failed to withdraw winnings.");
```

```
9
10 -         pendingWinnings[msg.sender] = 0;
11 -         emit WinningsWithdrawn(msg.sender, amount);
12       }
```

Informational

[I-1] `Game::initialGracePeriod` should be declared as immutable

Description The `Game::initialGracePeriod` is only assigned once during contract deployment in the `constructor` and is never modified afterward.

However, it is currently declared as a regular state variable, which consumes a storage slot and incurs higher gas costs for reads. Since this value remains constant throughout the contract's lifetime, it should be declared as **immutable** to optimize gas usage.

```
1 uint256 public initialGracePeriod;
```

Impact Unnecessary wastage of storage slot, which could be avoided.

Recommended mitigation

```
1 -         uint256 public initialGracePeriod;
2 +         uint256 public immutable i_initialGracePeriod;
```

Gas

[G-1] `feeIncreasePercentage` and `platformFeePercentage` percentage variables should use `uint8` instead of `uint256`

Description The contract uses `uint256` for percentage values that are expected to be `<= 100`. Both `feeIncreasePercentage` and `platformFeePercentage` represent percentage values that realistically will never exceed 100 **(representing 0-100%)**. Using `uint256` (32 bytes) for values that can be stored in `uint8` (1 byte) wastes storage slots and increases gas costs.

```
1 uint256 public feeIncreasePercentage; // Expected <= 100, could be uint8
2 uint256 public platformFeePercentage; // Expected <= 100, could be uint8
```

Each `uint256` consumes a full 32-**byte** storage slot, while `uint8` values can be packed together in a single slot when declared consecutively.

Impact Two full storage slots (64 bytes) used when one slot (32 bytes) could suffice

Recommended mitigation

```
-       uint256 public feeIncreasePercentage;
-       uint256 public platformFeePercentage;

+       uint8 public feeIncreasePercentage;
+       uint8 public platformFeePercentage;
```

[G-2] `getRemainingTime` and `getContractBalance` functions should be declared as external

Description Functions not called within the contract should be marked as external. External functions consume less gas as compared to public functions.

Impact Expensive to call public functions.

Recommended mitigation

```
-       function getRemainingTime() public view returns (uint256) {
+       function getRemainingTime() external view returns (uint256) {
                if (gameEnded) {
                    return 0; // Game has ended, no remaining time
                }
                uint256 endTime = lastClaimTime + gracePeriod;
                if (block.timestamp >= endTime) {
                    return 0; // Grace period has expired
                }
                return endTime - block.timestamp;
            }

-       function getContractBalance() public view returns (uint256) {
+       function getContractBalance() external view returns (uint256) {
                return address(this).balance;
            }
```