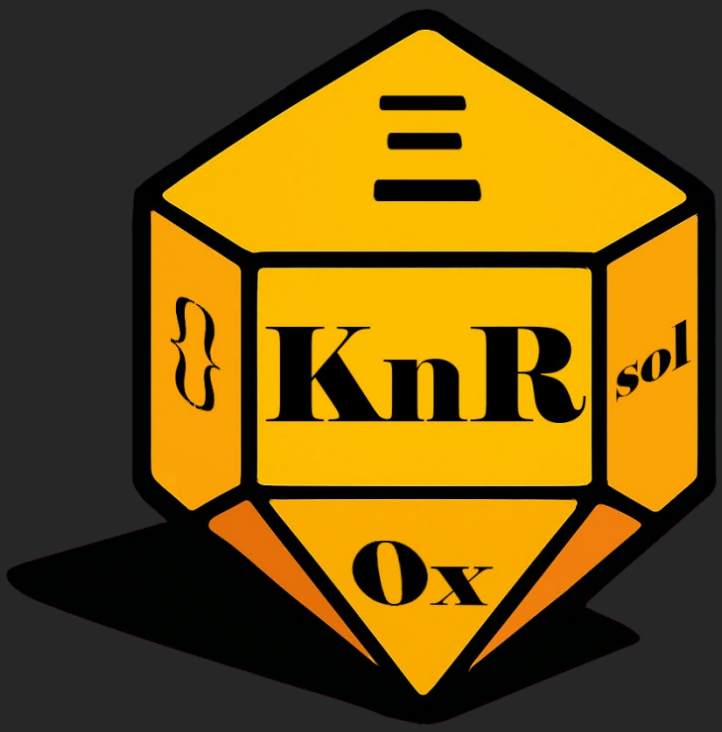

Kode-n-Rolla

Last Man Standing Security Review

Jul 31st 2025 - Aug 7th 2025

Table of Contents

Whoami

I am Pavel (aka kode-n-rolla), an independent security researcher.

Driven by a passion for decentralization and security, I focus on strengthening blockchain ecosystems.

I believe that security is an ongoing process, and my work contributes to making systems more resilient and reliable.

Disclaimer

This report reflects a time-boxed security review performed as part of a ***CodeHawks*** audit contest.

The analysis focuses exclusively on the security aspects of the reviewed ***Solidity*** codebase and does not guarantee the discovery of all potential vulnerabilities.

Findings represent the issues identified during the allotted review period and should not be considered an endorsement of the underlying protocol, business model, or operational security practices.

Due to the time-boxed and competitive nature of the review, this assessment does not guarantee the identification of all existing vulnerabilities. The absence of reported issues should not be interpreted as a claim of full security or correctness. Additional reviews, independent audits, and ongoing testing are recommended to improve the protocol's overall security posture.

Risk Classification

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

I use *CodeHawks* severity matrix to determine severity.

Impact refers to the potential harm or consequence to the users or the protocol due to the vulnerability.

- *High Impact*:
  - Funds are directly or nearly directly at risk
  - There's a severe disruption of protocol functionality or availability.
- *Medium Impact*:
  - Funds are indirectly at risk.
  - There's some level of disruption to the protocol's functionality or availability.
- *Low Impact*:
  - Funds are not at risk.
  - However, a function might be incorrect, the state might not be handled appropriately, etc.

Protocol Summary

The Last Man Standing protocol implements an on-chain competitive game inspired by the *"King of the Hill"* mechanic. Deployed as a Solidity smart contract on the Ethereum Virtual Machine (EVM), it allows participants to compete for a temporary "King" position by paying a progressively increasing entry cost.

Each successful claim resets a predefined grace period. If the grace period elapses without a new challenger, the current King is declared the winner and becomes entitled to withdraw the entire accumulated prize pool held by the contract.

Audit Details

This audit was conducted as part of a time-boxed competitive security review hosted on the CodeHawks platform. The assessment focused exclusively on the Solidity smart contract logic and its adherence to intended game mechanics, security assumptions, and fund safety.

The review did not include off-chain components, frontend integrations, or economic modeling beyond what is explicitly enforced by the contract code.

The review was performed during the "Last Man Standing" CodeHawks audit event.

Scope

The audit scope consisted of the following smart contract:

```
src/
└── Game.sol
```

Roles

The protocol defines the following actors and permissions:

Owner (Deployer)

Capabilities:

- Deploys the Game contract.
- Updates core game parameters, including *gracePeriod*, *initialClaimFee*, *feeIncreasePercentage*, and *platformFeePercentage*.
- Calls *resetGame()* to initialize a new round after a winner has been declared.
- Withdraws accumulated platform fees via *withdrawPlatformFees()*.

Restrictions:

- Cannot claim the throne while already assigned as the current King.
- Cannot declare a winner before the grace period has elapsed.
- Cannot reset the game while an active round is still in progress.

King (Current King)

Capabilities:

- Holds the King role as the most recent successful caller of *claimThrone()*.
- Receives a partial payout from subsequent throne claims, where applicable.
- Becomes eligible to withdraw the entire prize pool if no new claims occur before the grace period expires.
- Calls *withdrawWinnings()* once declared the winner.

Restrictions:

- Must pay the current *claimFee* to obtain the King role.
- Cannot reclaim the throne while already holding the King position.
- Can be displaced at any time by another participant submitting a valid claim.

Players (Claimants)

Capabilities:

- Call *claimThrone()* by supplying the required claim fee.
- Become the current King upon a successful claim.
- Potentially win the prize pool if they remain King until the grace period expires.

Restrictions:

- Must send sufficient ETH to meet or exceed the current *claimFee*.
- Cannot claim the throne after the game has concluded.

- Cannot claim the throne if already assigned as the current King.

Anyone (Winner Declarer)

Capabilities:

- Can call *declareWinner*() after the grace period has elapsed.

Restrictions:

- Cannot declare a winner before the grace period expires.
- Cannot declare a winner if no throne claims have occurred.
- Cannot declare a winner if the game has already concluded.

Executive Summary

This security review assessed the core game logic of the Last Man Standing smart contract, with a focus on access control, state transitions, fund distribution, and event correctness.

The review identified two medium-severity issues and one low-severity issue. The most impactful finding prevents any player from successfully claiming the throne due to an incorrect access control condition, effectively rendering the game non-functional. A second medium-severity issue breaks the intended reward mechanism for the previous King, causing deviations from the documented game mechanics and unfair player incentives. Additionally, a low-severity issue was identified where an emitted event reports incorrect prize information, leading to inconsistencies for off-chain consumers.

While no direct theft of funds was identified, the issues found can result in a complete denial of gameplay, broken incentive alignment, and misleading off-chain representations of game outcomes. Overall, the protocol's intended mechanics are not fully enforced at the contract level, and several logic assumptions require correction to ensure correct and reliable operation.

Issue found

Severity	Issue
Medium	Incorrect access control in *claimThrone()* prevents all gameplay
Medium	Previous King never receives intended reward payout
Low	*GameEnded* event emits incorrect prize amount

Findings

Medium

[M-01] Incorrect Access Control in claimThrone() Permanently Blocks Game Progression

Description

The *claimThrone()* function is responsible for allowing new participants to claim the King role by submitting the required *claimFee*. Under normal operation, any address other than the current King should be able to successfully claim the throne, triggering state updates and fee distribution.

However, the function contains an inverted access control condition that requires the caller to already be the current King. Since the initial value of *currentKing* is *address(0)*, and external callers can never satisfy this condition, all calls to *claimThrone()* revert unconditionally.

As a result, the core gameplay mechanic is rendered entirely non-functional from deployment.

```
function claimThrone() external payable gameNotEnded nonReentrant {
    require(
@>      msg.sender == currentKing, // Logic flaw: Should be `!=`
        "Game: You are already the king. No need to re-claim."
    );
    // ...
}
```

Impact (medium)

The protocol's primary functionality is completely disabled:

- No participant can ever successfully claim the throne.
- The game cannot progress beyond its initial state.
- No King is assigned, and no round can be completed.

Although no direct loss of funds occurs, the contract is effectively bricked, preventing any meaningful interaction and undermining the protocol's intended behavior.

Likelihood

High (Certain)

- The issue is deterministic and affects every invocation of *claimThrone()*.
- The failure occurs on the very first user interaction.
- No edge cases or special conditions are required to trigger the issue.

Proof of Concept

A Foundry-based test demonstrates that all calls to *claimThrone()* revert, even when sufficient ETH is supplied. The *currentKing* state variable remains *address(0)* indefinitely, confirming that no participant can ever assume the King role.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {Test, console2} from "forge-std/Test.sol";
import {Game} from "../src/Game.sol";

contract GameTest {
    Game public game;

    address public deployer;
    address public player1;
    address public player2;
    address public player3;
    address public maliciousActor;

    // Initial game parameters for testing
    uint256 public constant INITIAL_CLAIM_FEE = 0.1 ether; // 0.1 ETH
    uint256 public constant GRACE_PERIOD = 1 days; // 1 day in seconds
    uint256 public constant FEE_INCREASE_PERCENTAGE = 10; // 10%
    uint256 public constant PLATFORM_FEE_PERCENTAGE = 5; // 5%

    function setUp() public {
        deployer = makeAddr("deployer");
        player1 = makeAddr("player1");
        player2 = makeAddr("player2");
        player3 = makeAddr("player3");
        maliciousActor = makeAddr("maliciousActor");

        vm.deal(deployer, 10 ether);
        vm.deal(player1, 10 ether);
        vm.deal(player2, 10 ether);
        vm.deal(player3, 10 ether);
        vm.deal(maliciousActor, 10 ether);

        vm.startPrank(deployer);
        game = new Game(
            INITIAL_CLAIM_FEE,
            GRACE_PERIOD,
            FEE_INCREASE_PERCENTAGE,
            PLATFORM_FEE_PERCENTAGE
        );
        vm.stopPrank();
    }
    function test_ClaimThroneReverts() public {
        address testUser = player1; // any EOA attempting to claim the throne
        vm.deal(testUser, 1 ether);
        vm.prank(testUser);
        vm.expectRevert("Game: You are already the king. No need to re-claim.");
        game.claimThrone{value: 1 ether}();

        console2.log("Current King: ", game.currentKing());

        assertEq(game.currentKing(), address(0));
    }
```

Recommended Mitigation

Update the access control condition to prevent the current King from re-claiming the throne, rather than restricting claims exclusively to the current King:

```
function claimThrone() external payable gameNotEnded nonReentrant {
    require(
-       msg.sender == currentKing,
+       msg.sender != currentKing,
        "Game: You are already the king. No need to re-claim."
    );
    // ...
}
```

Additionally, consider explicitly handling the *currentKing == address(0)* case in payout logic to avoid unintended behavior during the first successful claim.

[M-02] Previous King Reward Is Never Paid Due to Uninitialized Payout Logic in *claimThrone()*

Description

According to the intended game mechanics and accompanying documentation, when a new player successfully claims the throne, a portion of the *claimFee* should be transferred to the previous King as a reward. This incentive is fundamental to the "King of the Hill" design, encouraging participation and competition.

However, within *claimThrone()*, the variable responsible for tracking this reward (*previousKingPayout*) is declared but never assigned a non-zero value. As a result, it remains 0 in all execution paths. Although subsequent logic references this variable defensively, no calculation or transfer to the previous King ever occurs.

This causes the reward mechanism for the displaced King to silently fail, despite being explicitly described as part of the protocol's behavior.

```
function claimThrone() external payable gameNotEnded nonReentrant {
    ...
@>  uint256 previousKingPayout = 0; // Declared but never updated
    ...
    // Defensive check referencing previousKingPayout
    if (currentPlatformFee > (sentAmount - previousKingPayout)) {
        currentPlatformFee = sentAmount - previousKingPayout;
    }
    ...
    // No transfer to previous king anywhere
}
```

Impact (medium)

The intended incentive structure of the game is broken:

- Previous Kings never receive compensation when they are dethroned.
- The "King of the Hill" reward mechanism is effectively disabled.
- Player expectations, as set by documentation and code comments, are not met.

While this issue does not directly compromise fund safety, it results in unfair gameplay outcomes and undermines the protocol's core economic design.

Likelihood

High

- The issue affects every *claimThrone()* call following the initial claim.
- Once a King exists, the payout logic is expected to execute but always results in a zero-value reward.
- No special conditions or edge cases are required to trigger the issue.

Proof of Concept

A local test demonstrates that after a King is dethroned via *claimThrone()*, the previous King's balance remains unchanged, confirming that no reward payout is ever executed.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {Test, console2} from "forge-std/Test.sol";
import {Game} from "../src/Game.sol";

contract GameTest is Test {
    Game public game;

    address public deployer;
    address public player1;
    address public player2;
    address public player3;
    address public maliciousActor;

    // Initial game parameters for testing
    uint256 public constant INITIAL_CLAIM_FEE = 0.1 ether; // 0.1 ETH
    uint256 public constant GRACE_PERIOD = 1 days; // 1 day in seconds
    uint256 public constant FEE_INCREASE_PERCENTAGE = 10; // 10%
    uint256 public constant PLATFORM_FEE_PERCENTAGE = 5; // 5%

    function setUp() public {
        deployer = makeAddr("deployer");
        player1 = makeAddr("player1");
        player2 = makeAddr("player2");
        player3 = makeAddr("player3");
        maliciousActor = makeAddr("maliciousActor");

        vm.deal(deployer, 10 ether);
        vm.deal(player1, 10 ether);
        vm.deal(player2, 10 ether);
        vm.deal(player3, 10 ether);
        vm.deal(maliciousActor, 10 ether);

        vm.startPrank(deployer);
        game = new Game(
            INITIAL_CLAIM_FEE,
            GRACE_PERIOD,
            FEE_INCREASE_PERCENTAGE,
            PLATFORM_FEE_PERCENTAGE
        );
        vm.stopPrank();
    }

    function test_PreviousKinkDoesNotReceivePayout() public {
        // claimThrone with player1
        vm.deal(player1, 1 ether);
        vm.startPrank(player1);
        game.claimThrone{value: 1 ether}();
        vm.stopPrank();

        // Balance visability before
        console2.log(
            "Player1 balance after claiming throne: ",
            player1.balance
```

```
        );

        // claimThrone with player2
        vm.deal(player2, 2 ether);
        vm.startPrank(player2);
        game.claimThrone{value: 2 ether}();
        vm.stopPrank();

        // Balance visability after
        console2.log(
            "Player1 balance after player2 claims throne: ",
            player1.balance
        );

        assertEq(
            player1.balance,
            0,
            "Player1 should have received payout but didn't"
        );
    }
}
```

Recommended Mitigation

Ensure that previousKingPayout is correctly calculated based on the intended reward logic and that the computed amount is explicitly transferred to the previous King during a successful throne claim.

```
...
// Game Core State
...
+   uint256 previousKingRewardPercentage = 10; // For calculate previuosKinh
payout
...

 function claimThrone() external payable gameNotEnded nonReentrant {
...

     uint256 sentAmount = msg.value;

-    uint256 previousKingPayout = 0;
-    uint256 currentPlatformFee = 0;
-    uint256 amountToPot = 0;
+    // Payout to the previous king
+    uint256 previousKingPayout = (sentAmount * previousKingRewardPercentage) /
100;
+
+    if (currentKing != address(0) && previousKingPayout > 0) {
+        (bool sent, ) = currentKing.call{value: previousKingPayout}("");
+        require(sent, "Failed to send payout to previous king");
+    }
```

```
+
+     // Calculate platform fee and amount added to the pot
+     uint256 remaining = sentAmount - previousKingPayout;
+     uint256 currentPlatformFee = (remaining * platformFeePercentage) / 100;
+
+     // Safety check: fee should not exceed the remaining amount
+     if (currentPlatformFee > remaining) {
+         currentPlatformFee = remaining;
+     }
+     platformFeesBalance += currentPlatformFee;
+
+     uint256 amountToPot = remaining - currentPlatformFee;
+     pot += amountToPot;

-     // Calculate platform fee
-     currentPlatformFee = (sentAmount * platformFeePercentage) / 100;
-
-     // Defensive check to ensure platformFee doesn't exceed available amount
after previousKingPayout
-     if (currentPlatformFee > (sentAmount - previousKingPayout)) {
-         currentPlatformFee = sentAmount - previousKingPayout;
-     }
-     platformFeesBalance = platformFeesBalance + currentPlatformFee;
-
-     // Remaining amount goes to the pot
-     amountToPot = sentAmount - currentPlatformFee;
-     pot = pot + amountToPot;

      // Update game state
      currentKing = msg.sender;
...
```

Low

[L-01] GameEnded Event Emits Zero Prize Amount Due to State Reset Before Emission

Description

When *declareWinner()* finalizes a round, the winner's prize (stored in pot) is credited to *pendingWinnings*, and the *GameEnded* event is emitted to announce the outcome. Off-chain consumers (frontends, indexers, explorers) may rely on this event to display the winning amount and build historical records.

However, the contract resets pot to zero before emitting the *GameEnded* event. As a result, the event's *prizeAmount*

parameter is always emitted as 0, even when the winner is credited with a non-zero prize internally.

```
pot = 0; // // Reset pot after assigning to winner's pending winnings

emit GameEnded(currentKing, pot, block.timestamp, gameRound); // @> emits zero value instead of actual pot
```

This creates a mismatch between the event logs and the contract's actual accounting state.

Impact

- Off-chain systems relying on *GameEnded* will consistently display an incorrect prize amount (0).
- Analytics and historical tracking may be inaccurate unless they perform additional state reads.
- Integrations that trust emitted data (rather than reconstructing state transitions) may present misleading results to users.

This issue does not affect fund safety or on-chain accounting, but it degrades observability and ecosystem compatibility.

Likelihood

- The issue occurs deterministically on every successful call to *declareWinner()*.

- No special conditions are required beyond reaching the normal winner-declaration flow.

Proof of Concept

1. Deploy Game with valid parameters.
2. Call *claimThrone()* with sufficient ETH to create a non-zero pot.
3. Advance time beyond *gracePeriod*.
4. Call *declareWinner()*.
5. Observe that *pendingWinnings[currentKing]* increases by the expected prize amount, while the emitted *GameEnded* event reports *prizeAmount = 0*.

```
// Signature of the event to observe
event GameEnded(address indexed winner, uint256 prizeAmount, uint256 timestamp,
uint256 round);
```

Recommended Mitigation

Emit the GameEnded event before resetting pot, or cache the pot value in a local variable and emit the cached value:

- Store *uint256 prize = pot;*
- Credit *pendingWinnings using prize*
- *Emit GameEnded(..., prize, ...)*
- Reset *pot* after emission

This ensures that event logs accurately reflect the actual prize awarded.

```
function declareWinner() external gameNotEnded {
    require(currentKing != address(0), "Game: No one has claimed the throne
yet.");
    require(
        block.timestamp > lastClaimTime + gracePeriod,
        "Game: Grace period has not expired yet."
    );

    gameEnded = true;

    pendingWinnings[currentKing] = pendingWinnings[currentKing] + pot;
+
+   emit GameEnded(currentKing, pot, block.timestamp, gameRound);
+
    pot = 0;
-
-   emit GameEnded(currentKing, pot, block.timestamp, gameRound);
}
);
```