Last Man Standing Competitive Audit Report

Version 1.0

Cyfrin.io

August 8, 2025

Last Man Standing Competitive Audit Report

Mudassir

August 8, 2025

Prepared by: Mudassir Lead Auditors: - Mudassir

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

Mudassir makes all effort to find as many vulnerabilities in the code in the given time period, NOTE: this is the Report of a Competitive Audit (firstFlight) from CodeHawks.

Risk Classification

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

Issues found

Severity	Number of issues found
High	2
Medium	0
Low	2
Info	1
Gas Optimizations	0
Total	5

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

Scope

‘Game.sol’

Findings

Highs

[H-1] Centralization Risk!

Description: Centralization Risk, Owner can set the Fees given to him or the platform as much as he wants to!!!.

The Owner has too much Control of the Contract thus risking Decentralization. People are not going to be very happy seeing Owner with too much access.

Impact: HIGH

Proof of Concept: Owner sets the platform fee to 100%

David tries to claim the throne with 1 ETH

All of David's 1 ETH goes to platformFeesBalance (owner), nothing to pot.

Additionally Owner set initialClaimFee to 100 ether or watever he wants.

No one can afford to play anymore. Game is locked.

Recommended Mitigation: Set the Fee amount changed by Owner cannot exceed a threshold in Constant State Variables and Create a Modifier that Restricts the Owner.in the code below and actually the overall code implement a Mechanism that the Owner cannot set the Fees too high, and restrict Owner so that he can only do what s need to be done.

[H-2] Incorrect Check in 'ClaimThrone' function prevents anyone else from Claiming the Throne thus breaking Functionality

Description: Functionality Broken. The 'ClaimThrone' function has a check that prevents anyone else from Claiming the Throne. Impact : High, as the first Throne Claimer becomes the King forever as noone can Claim the Throne. Description The Require Check in 'ClaimOwner' checks the msg.sender is equal to the currentKing it reverts as the new throne claimer is ofc not the King.

Impact: HIGH

Likelihood HIGH

Proof of Concept: Account 1 becomes the King he is the first King since deployment

Account 2 tries to become the king by claimingThrone with the amount greater then the previousAmount but it reverts with error message "You are already the King" While Hes Not!!!

Cause its saying that the msg.sender is equal to currentKing, So No one else can become the king as they aren't the current King.

To demonstrate this issue, I wrote a test where UserA becomes the king, and UserB attempts to claim the throne. The call reverts, even though UserB is not the current king — this is clearly unintended behavior and shows that the logic is flipped

[46399] GameTest::testNewUserCannotClaim_IfRequireBugged() [0] VM::prank(UserB: [0x7E5F4552091A69125d5DfCb7b
[Return] [29259] Game::claimThrone{value: 100000000000000000}() storage changes: @ 16: 0 → 1 [Revert] Game: You are already the king. No need to re-claim. [Revert] Game: You are already the king. No need to re-claim.

Recommended Mitigation:

```
- 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.");
```

Lows

[L-1] Literal Values/ Magic Numbers are being used throughout the Contract

Description: The Game.sol Contract uses hardcoded literals which is not a good practice Literal Values/ Magic Numbers are being used throughout the Contract, a good practice would be to declare a Constant Variable one time in the contract and use it throughout the Contract Decrease readability, especially in large contracts. Make the code harder to maintain or refactor.

Recommended Mitigation:

```
+  SECONDS_IN_A_DAY
```

[L-2] Event not returning the correct amount sent to the winner

Description: The Event is not emitting correct info rather its emitting an 0 for that variable and it will Emit 0 all the time because we are getting that value which will be 0 , thus not benefitting us ,though ofc its not beneficial as it just returns 0,what we can do in place of this is that we can store the amount given to the player in a other variable and then Emit that here.

Recommended Mitigation:

```
+   uint256 prizeAmount = pot; , // this line should be added to emit
   in the event that how much the winner got the Amount
```