Last Man Standing Audit Report

Version 1.0

ChargingFoxSec

August 8, 2025

Last Man Standing Audit Report

ChargingFoxSec

August 7, 2025

Prepared by:
ChargingFoxSec

Lead Auditor:
ChargingFoxSec

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 ChargingFoxSec 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

```
src/Game.sol
```

Roles

This protocol includes the following roles:

1. Owner (Deployer)
2. King (Current King)
3. Players (Claimants)
4. Anyone (Declarer)

Executive Summary

We spent a few hours performing static analysis, fuzz testing, and invariant testing to identify the following issues.

Issues found

severity	number
High	4
Medium	0
Low	0
Informational	1
Gas	1
total	6

Findings

High

[H-1]Use of block.timestamp Enables Game Result Manipulation by Miners

Description

- The contract use block.timestamp to calculate lastClaimTime + gracePeriod, and check if can call the declareWinner().
- Miners can control the block.timestamp in some range(±15 seconds).That might influence the condition of calling declareWinner().In some extreme condition , the attacker would win the game unfairly by manipulate the block.timestamp.

```
// Root cause in the codebase with @> marks to highlight the relevant section
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;
        pot = 0; // Reset pot after assigning to winner's pending winnings

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

Risk

Likelihood:

- When the gracePeriod is near the miners control time window, malicious miner who is also a player can influence the result by modifying the block.timestamp.

Impact:

- A malicious miner-player can unfairly claim the prize, depriving other participants.

Proof of Concept

- A miner who is also a player (e.g. Alice) notices that she is the currentKing and that the condition block.timestamp > lastClaimTime + gracePeriod will soon be met within a few seconds.
- By adjusting the block's timestamp to T + gracePeriod + X (where X is within the miner's ±15-second manipulation range), she forces the condition to succeed.
- In that same block, she calls declareWinner(), prematurely ending the game and securing the victory.
- **Result**: Alice unfairly ends the game early and claims the prize, denying other players a fair chance.
- Impact example:https://www.cyfrin.io/glossary/block-timestamp-manipulation-hack-solidity-code-example

Recommended Mitigation

- Use the block.number instead of block.timestamp for comparing.

```
// Game Core State
-    uint256 public gracePeriod; // Time in seconds after which a winner can be declared (e.g., 24 hours)
+   uint256 public gracePeriod; // Block increase number after which a winner can be declared(e.g., 100 blocks)
-    uint256 public lastClaimTime; // Timestamp when the throne was last claimed
+    uint256 public lastClaimBlockNumber; // block number which the throne was last claimed
```

```
constructor(
        uint256 _initialClaimFee,
        uint256 _gracePeriod,
        uint256 _feeIncreasePercentage,
        uint256 _platformFeePercentage
    ) Ownable(msg.sender) {
        // Set deployer as owner
```

```
        require(
            _initialClaimFee > 0,
            "Game: Initial claim fee must be greater than zero."
        );
        require(
            _gracePeriod > 0,
            "Game: Grace period must be greater than zero."
        );
        require(
            _feeIncreasePercentage <= 100,
            "Game: Fee increase percentage must be 0-100."
        );
        require(
            _platformFeePercentage <= 100,
            "Game: Platform fee percentage must be 0-100."
        );

        initialClaimFee = _initialClaimFee;
        initialGracePeriod = _gracePeriod;
        feeIncreasePercentage = _feeIncreasePercentage;
        platformFeePercentage = _platformFeePercentage;

        // Initialize game state for the first round
        claimFee = initialClaimFee;
        gracePeriod = initialGracePeriod;
-        lastClaimTime = block.timestamp; // Game starts immediately upon deployment
+       lastClaimBlockNumber = block.number;//Game starts immediately upon deployment
        gameRound = 1;
        gameEnded = false;
        // currentKing starts as address(0) until first claim
    }
```

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

        gameEnded = true;

        pendingWinnings[currentKing] = pendingWinnings[currentKing] + pot;
        pot = 0; // Reset pot after assigning to winner's pending winnings
```

```
16
17          emit GameEnded(currentKing, pot, block.timestamp, gameRound);
18      }
```

```
1  function resetGame() external onlyOwner gameEndedOnly {
2          currentKing = address(0);
3  -       lastClaimTime = block.timestamp;
4  +       lastClaimBlockNumber = block.number;
5          pot = 0;
6          claimFee = initialClaimFee;
7          gracePeriod = initialGracePeriod;
8          gameEnded = false;
9          gameRound = gameRound + 1;
10         // totalClaims is cumulative across rounds, not reset here, but
               could be if desired.
11
12         emit GameReset(gameRound, block.timestamp);
13     }
```

```
1  -function getRemainingTime() public view returns (uint256) {
2  -        if (gameEnded) {
3  -            return 0; // Game has ended, no remaining time
4  -        }
5  -        uint256 endTime = lastClaimTime + gracePeriod;
6  -        if (block.timestamp >= endTime) {
7  -            return 0; // Grace period has expired
8  -        }
9  -        return endTime - block.timestamp;
10 -    }
11 +function getRemainingBlockNumber() public view returns (uint256) {
12 +        if (gameEnded) {
13 +            return 0; // Game has ended, no remaining time
14 +        }
15 +        uint256 endBlockNumber = lastClaimBlockNumber + gracePeriod;
16 +        if (block.number >= endBlockNumber) {
17 +            return 0; // Grace period has expired
18 +        }
19 +        return endBlockNumber - block.number;
20 +    }
```

[H-2]Missing Address Type Check Before Low-level Call May Lead To Malicious Code Execution

Description

- The contract uses payable(address).call{value: amount}(“”) to transfer ETH to user-controlled addresses in withdrawWinnings() and withdrawPlatformFees().
- If the target address is not an Externally Owned Account (EOA) but a contract, the low-level call will invoke the contract’s fallback() or receive() function, which may contain malicious logic such as re-entrancy.

```
1
2  function withdrawWinnings() external nonReentrant {
3          uint256 amount = pendingWinnings[msg.sender];
4          require(amount > 0, "Game: No winnings to withdraw.");
5
6  @>         (bool success, ) = payable(msg.sender).call{value: amount}("");
7          require(success, "Game: Failed to withdraw winnings.");
8
9          pendingWinnings[msg.sender] = 0;
10
11         emit WinningsWithdrawn(msg.sender, amount);
12     }
13 function withdrawPlatformFees() external onlyOwner nonReentrant {
14         uint256 amount = platformFeesBalance;
15         require(amount > 0, "Game: No platform fees to withdraw.");
16
17         platformFeesBalance = 0;
18
19 @>         (bool success, ) = payable(owner()).call{value: amount}("");
20         require(success, "Game: Failed to withdraw platform fees.");
21
22         emit PlatformFeesWithdrawn(owner(), amount);
23     }
```

Risk

Likelihood:High

- These functions allow sending ETH to arbitrary addresses without checking whether the recipient is a contract.
- When the recipient is a contract, the fallback or receive function is triggered, allowing it to perform arbitrary logic, including re-entrant calls.

Impact:High

- Can cause the re-entrant issue(etc. take away all the ether of the Game contract).
- It could also cause cross-function re-entrancy, bypassing access control or logic assumptions elsewhere in the contract.
- **Note:** While nonReentrant provides protection against direct re-entrancy in the current context, relying solely on modifiers is not always sufficient.

> For example, if future changes introduce internal calls, delegatecalls, or access through another entry point, the low-level call may still become a re-entrancy vector.

> In addition, the fallback or receive function of the recipient may contain **side effects** that interfere with contract logic or gas usage, even without re-entrancy.

Proof of Concept

A malicious contract can exploit the low-level call as follows:

```
//fallback function in malicious contract
function fallback() external {
    address GameContractAddress = 0x....aaa;
    // Re-enter the Game contract
    address(GameContractAddress).withdrawWinnings();
}
```

In this example, when the Game contract sends ETH to the malicious contract, it triggers the fallback function, which re-enters the vulnerable withdrawWinnings() logic before the previous withdrawal finalizes.

Recommended Mitigation

- Option 1: Use transfer() or send() instead of call
  - These have fixed gas stipends and are safer for simple transfers (but less flexible).
  - Note: transfer() reverts on failure, while send() returns false.

```
payable(msg.sender).transfer(amount);
```

- Option 2: Validate recipient is an EOA before sending ETH
  - Use extcodesize to detect whether the address is a contract:

```
require(msg.sender.code.length == 0, "Game: recipient is a contract");
```

[H-3]withdrawWinnings() Does Not Follow the Checks-Effects-Interactions (CEI) Pattern, Leading to Potential Re-Entrancy Risk

Description

- In WithdrawWinning() ,the line that resets pendingWinnings[msg.sender] to 0 is placed after the external call.
- This violates the Checks-Effects-Interactions (CEI) pattern and introduces a potential re-entrancy vulnerability. If a malicious contract receives ETH via the low-level call, it could re-enter the withdrawWinnings() function before its pendingWinnings is cleared.
- As a result, the attacker can bypass the amount > 0 check and repeatedly drain the contract's balance.

```
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);
    }
```

Risk

Likelihood:High

- The function was calling the external call to another address,when the recipient is a contract, the fallback or receive function is triggered, allowing it to perform arbitrary logic, including re-entrant calls.

Impact:High

- Since the require(amount > 0) is useless , the attacker can repeatedly call the withdrawWinnings() till the balance of the Game contract is drained. so the attacker is successfully to steal all the ether which does not belong to him.
- Note: While nonReentrant provides protection against direct re-entrancy in the current context, relying solely on modifiers is not always sufficient.

> For example, if future changes introduce internal calls, delegatecalls, or access through another entry point, the low-level call may still become a re-entrancy vector.

> In addition, the fallback or receive function of the recipient may contain **side effects** that interfere with contract logic or gas usage, even without re-entrancy.

Proof of Concept

A malicious contract can exploit the low-level call as follows:

```
1 //fallback function in malicious contract
2 function fallback() external {
3     address GameContractAddress = 0x....aaa;
4     // Re-enter the Game contract
5     address(GameContractAddress).withdrawWinnings();
6 }
```

In this example, when the Game contract sends ETH to the malicious contract, it triggers the fallback function, which re-enters the vulnerable withdrawWinnings() logic before the previous withdrawal finalizes.And while the pendingWinnings[msg.sender] was not be set to zero, so it can still bypass the amount > 0 check until the contract's entire balance is drained.

Recommended Mitigation

- Following the Checks-Effects-Interactions (CEI) pattern will protect contracts from re-entrancy issues.
- Placed the line that resets the variable pendingWinner before the external call.

```
1 function withdrawWinnings() external nonReentrant {
2         uint256 amount = pendingWinnings[msg.sender];
3         require(amount > 0, "Game: No winnings to withdraw.");
4 +       pendingWinnings[msg.sender] = 0;
5         (bool success, ) = payable(msg.sender).call{value: amount}("");
```

```
6           require(success, "Game: Failed to withdraw winnings.");
7
8   -       pendingWinnings[msg.sender] = 0;
9
10          emit WinningsWithdrawn(msg.sender, amount);
11      }
```

[H-4]Condition Of claimThrone() Is Wrong,Leads To Nobody Can Claim The King

Description

- claimThrone() use require(msg.sender == currentKing) to check if the msg.sender is the currentKing.
- But the condition is wrong, that makes no player can continue the game.

```
1   function claimThrone() external payable gameNotEnded nonReentrant {
2           require(
3               msg.value >= claimFee,
4               "Game: Insufficient ETH sent to claim the throne."
5           );
6           require(
7   @>          msg.sender == currentKing,
8               "Game: You are already the king. No need to re-claim."
9           );
10
11          uint256 sentAmount = msg.value;
12          uint256 previousKingPayout = 0;
13          uint256 currentPlatformFee = 0;
14          uint256 amountToPot = 0;
15
16          // Calculate platform fee
17          currentPlatformFee = (sentAmount * platformFeePercentage) /
                100;
18
19          // Defensive check to ensure platformFee doesn't exceed
                available amount after previousKingPayout
20          if (currentPlatformFee > (sentAmount - previousKingPayout)) {
21              currentPlatformFee = sentAmount - previousKingPayout;
22          }
23          platformFeesBalance = platformFeesBalance + currentPlatformFee;
24
25          // Remaining amount goes to the pot
```

```
26          amountToPot = sentAmount - currentPlatformFee;
27          pot = pot + amountToPot;
28
29          // Update game state
30          currentKing = msg.sender;
31          lastClaimTime = block.timestamp;
32          playerClaimCount[msg.sender] = playerClaimCount[msg.sender] +
                1;
33          totalClaims = totalClaims + 1;
34
35          // Increase the claim fee for the next player
36          claimFee = claimFee + (claimFee * feeIncreasePercentage) / 100;
37
38          emit ThroneClaimed(
39              msg.sender,
40              sentAmount,
41              claimFee,
42              pot,
43              block.timestamp
44          );
45      }
```

Risk

Likelihood: High

- All the players will call this function, otherwise they will not win the game.But since no one can satisfy the condition require(msg.sender == currentKing), so that makes revert every time when player call this function.

Impact:High

- Nobody can win the game.
- Nobody can continue the game.

Proof of Concept

Since no player is the king in the first place.So claimThrone() will revert forever when players call.

```
1  require(
2              msg.sender == currentKing,
3              "Game: You are already the king. No need to re-claim."
4          );
```

Recommended Mitigation

- Replace the incorrect condition require(msg.sender == currentKing) with require(msg.sender != currentKing) to ensure that only new players (not the current king) can claim the throne.
- This prevents unnecessary reverts and allows the game to progress normally.

```
function claimThrone() external payable gameNotEnded nonReentrant {
        require(
            msg.value >= claimFee,
            "Game: Insufficient ETH sent to claim the throne."
        );
        require(
-            msg.sender == currentKing,
+            msg.sender != currentKing,
            "Game: You are already the king. No need to re-claim."
        );

        uint256 sentAmount = msg.value;
        uint256 previousKingPayout = 0;
        uint256 currentPlatformFee = 0;
        uint256 amountToPot = 0;

        // 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;
        lastClaimTime = block.timestamp;
        playerClaimCount[msg.sender] = playerClaimCount[msg.sender] +
            1;
        totalClaims = totalClaims + 1;

        // Increase the claim fee for the next player
        claimFee = claimFee + (claimFee * feeIncreasePercentage) / 100;

        emit ThroneClaimed(
            msg.sender,
            sentAmount,
```

```
42              claimFee,
43              pot,
44              block.timestamp
45          );
46      }
```

Informational

[I-1]pot Is Reset Before Emitting GameEnded, Causing Logged Value to Always Be 0

Description

- GameEnded event is emitted after declareWinner() is called.And it will log some information(ect. winner,pot).
- The value of pot in log will always be 0.

```
1  function declareWinner() external gameNotEnded {
2          require(
3              currentKing != address(0),
4              "Game: No one has claimed the throne yet."
5          );
6          require(
7              block.timestamp > lastClaimTime + gracePeriod,
8              "Game: Grace period has not expired yet."
9          );
10
11         gameEnded = true;
12
13         pendingWinnings[currentKing] = pendingWinnings[currentKing] +
               pot;
14 @>       pot = 0; // Reset pot after assigning to winner's pending
       winnings
15
16         emit GameEnded(currentKing, pot, block.timestamp, gameRound);
17     }
```

Risk

Likelihood:High

- It will happen when GameEnded event is emitted.

Impact:Low

- It does not cause any problem but log the value always 0 do not make any sense.

Proof of Concept

- GameEnded event emitted after variable pot set to 0.

```
    pot = 0; // Reset pot after assigning to winner's pending winnings
    emit GameEnded(currentKing, pot, block.timestamp, gameRound);
```

Recommended Mitigation

- Change the order of reset variable pot and emitted the GameEnded event would solve this problem.

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

        emit GameEnded(currentKing, pot, block.timestamp, gameRound);
+       pot = 0; // Reset pot after assigning to winner's pending
    winnings
    }
```

Gas

[G-1]Unchanged Variable Should Use constant or immutable

Description

- The variable initialGracePeriod is assigned once during contract deployment and never modified afterwards.However, it is declared as a regular public storage variable instead of using immutable or constant keyword.This leads to unnecessary storage usage and higher gas costs.

```
1 @>     uint256 public initialGracePeriod;
```

Risk

Likelihood:

- High
- Happens frequently in contracts during developing.

Impact:

- Low
- Cost more gas when deploying the contract and contract in runtime.

Proof of Concept

```
1 // current
2 uint256 public initialGracePeriod;
3 // better
4 uint256 public immutable initialGracePeriod;
```

Recommended Mitigation

Change the variable initialGracePeriod to immutable.

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