Contents

BlackJack Contract Security Review

Overview

This document contains the findings of the BlackJack smart contract security review. The contract implements a decentralized blackjack game where players can bet ETH and play against a dealer.

Severity Classification

- High: Vulnerabilities that can lead to direct loss of funds or complete contract failure
- Medium: Issues that pose indirect risks or significant maintenance concerns
- Low: Best practice violations and code quality issues

Findings Summary

- 2 High severity findings
- 2 Medium severity findings
- 1 Low severity finding

Detailed Findings

[H-1] Contract becomes insolvent due to missing initial deposit mechanism

Description: The contract is designed to pay out 2 ETH for winning games but only accepts 1 ETH deposits through startGame(). Without additional ETH funding mechanisms (like receive() or payable constructor), the contract will eventually become insolvent and unable to pay winners.

Impact: - Contract will fail to pay winners once balance drops below 2 ETH - Players could lose their 1 ETH deposits without possibility of winning - Complete failure of core contract functionality

Proof of Concept: 1. Players deposit 1 ETH each 2. Winners should receive 2 ETH 3. After several winning games, contract balance becomes insufficient 4. New winners cannot receive their rewards

Recommended Mitigation: Add funding mechanism through constructor and receive function:

```
+    constructor() payable {}
+    receive() external payable {}
```

[H-2] Incorrect game winner determination logic leads to unfair payouts

Description: The call() function has several logical flaws in determining the winner of the game. It doesn't properly check for player busts and has incorrect win/loss conditions.

Impact: - Players can win even when bust (over 21) - Incorrect payouts possible - Potential for unfair wins/losses - Direct financial impact on players

Proof of Concept:

```
if (dealerHand > 21) {
    emit PlayerWonTheGame(
        "Dealer went bust, players winning hand: ",
        playerHand
    );
    endGame(msg.sender, true);
} else if (playerHand > dealerHand) {
    emit PlayerWonTheGame(
        "Dealer's hand is lower, players winning hand: ",
        playerHand
    );
    endGame(msg.sender, true);
} else {
    emit PlayerLostTheGame(
        "Dealer's hand is higher, dealers winning hand: ",
        dealerHand
    );
    endGame(msg.sender, false);
}
```

Recommended Mitigation: Add proper bust checks and game logic:

```
+   if (playerHand > 21) { // Better to use constant
+       emit PlayerLostTheGame("Player is bust", playerHand);
+       endGame(msg.sender, false);
+       return;
+   }
```

[M-1] Missing owner withdrawal mechanism creates fund management risks

Description: The contract lacks both ownership functionality and withdrawal mechanisms. There is no way to withdraw accumulated funds from the contract, and no designated owner who could manage the contract's operations.

Impact: - ETH can become permanently locked in the contract - No administrative control over contract operations - No way to handle emergency situations

Recommended Mitigation: 1. Add ownership mechanism:

```
constructor() payable {
    owner = msg.sender;
}

modifier onlyOwner() {
```

```
    require(msg.sender == owner, "Not the owner");
    _;
}
```

2. Add withdraw function:

```
function withdrawFees(uint256 amount) external onlyOwner {
    require(amount <= address(this).balance, "Insufficient balance");
    require(address(this).balance - amount >= 2 ether, "Must keep minimum balance for game
    (bool success, ) = payable(owner).call{value: amount}("");
    require(success, "Transfer failed");
    emit FeeWithdrawn(owner, amount);
}
```

[M-2] Excessive use of magic numbers reduces code maintainability

Description: The contract contains multiple hardcoded numeric values without clear documentation or constant definitions. These numbers are used for game logic, card calculations, and ETH values.

Impact: - Reduced code readability - Higher maintenance difficulty - Increased risk of errors during updates - Lack of clarity in business logic

Proof of Concept:

```
for (uint256 i = 1; i <= 52; i++) {  // Magic number: 52 cards
    availableCards[player].push(i);
}

uint256 standThreshold = (randomValue % 5) + 17;  // Magic numbers: 5, 17

if (dealerHand > 21) {  // Magic number: 21
    // ...
}

payable(player).transfer(2 ether);  // Magic number: 2 ether
```

Recommended Mitigation: Define constants for all magic numbers:

```
+   uint256 private constant DECK_SIZE = 52;
+   uint256 private constant BLACKJACK_VALUE = 21;
+   uint256 private constant MIN_DEALER_STAND = 17;
+   uint256 private constant DEALER_STAND_RANGE = 5;

    // Payment constants
+   uint256 private constant REQUIRED_BET = 1 ether;
+   uint256 private constant WINNING_PAYOUT = 2 ether;

    // Card constants
+   uint256 private constant FACE_CARD_VALUE = 10;
+   uint256 private constant CARDS_PER_SUIT = 13;
```

[L-1] Non-specific Solidity version poses compatibility risks

Description: The contract uses a floating pragma (^0.8.13), which allows compilation with any version equal to or greater than 0.8.13.

Impact: - Potential for inconsistent behavior across deployments - Different compiler versions might introduce varying optimizations - Security fixes in newer versions might not be utilized

Recommended Mitigation: Use a specific version:

```
- pragma solidity ^0.8.13;
+ pragma solidity 0.8.28;
```

Conclusion

The contract contains several critical issues that need to be addressed before deployment. The most urgent concerns are the insolvency risk and incorrect game logic, which could directly impact user funds. Medium severity issues around contract management and code maintainability should also be addressed to ensure long-term sustainability.