Protocol Audit Report

Version 1.0

1may Hawks

November 24, 2024

Example Protocol Audit Report

Georgi Karov

November 24, 2024

Prepared by: 1may Hawks Lead Security Researchers:

- Georgi Karov

Table of Contents

Protocol Summary

The “TwentyOne” protocol is a smart contract implementation of the classic blackjack card game, where users can wager 1 ETH to participate with a chance to double their money! More information can be found in CodeHawks

Disclaimer

The 1may Hawks Security 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 review (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
Likelihood	High	H	H/M	M
	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

This is a security review of the CodeHawks first flight TwentyOne contest. Even though it is a competitive contest I wanna practice my report-generation skills, so I have decided to make my findings into an example report.

Scope

All Contracts in `src` are in scope. src/TwentyOne.sol ## Roles Player: The user who interacts with the contract to start and play a game. A player must deposit 1 ETH to play, with a maximum payout of 2 ETH upon winning.

```
1 Dealer: The virtual counterpart managed by the smart contract. The
      dealer draws cards based on game logic.
```

Executive Summary

Codehawks makes the so called first flight competitive audit contests that are a simplified replica of a real contest and have no real money prize pools, but serve as a good opportunity for beginner security researchers to hone their skills and prepare for real competitive contests like the ones on CodeHawks, Code4rena, Sherlock and similar platforms.

Issues found

The 1may Hawks Security team managed to find 5 High severity vulnerabilities, 1 Medium severity vulnerability and 1 Low severity vulnerability, which contains 9 more gas and imformatioanal improvements that were suggested to the protocol by the 1may Hawks.

Findings

High

[H-1] No Good Way to Fund the TwentyOne Game Contract Initially or Withdraw Funds

Summary

The contract does not support a `payable` constructor, `receive()`, or `fallback()` functions, rendering it essentially unplayable. Additionally, there is no mechanism to withdraw funds from the contract, causing any forfeited user Ether to remain permanently stuck.

Vulnerability Details

The contract lacks a straightforward way for the owner to fund the initial prize pool or withdraw rewards. While some workarounds exist, they are not intuitive, not documented as expected behavior, and may lead to undesirable consequences.

Potential Workarounds

1. **Using `startGame()` with Sufficient ETH:**
   The owner could deploy the contract with no initial balance and fund it later by calling the `startGame()` function with enough Ether. However:
   - This approach requires unnecessary transactions.
   - It starts a game for the owner, which is not ideal.
   - There is no guarantee that another player will not have already played the game.
2. **Deploying to a Pre-Funded Address:**
   The owner could deploy the contract to a predetermined address with existing ETH. However, this:
   - Requires advanced techniques.
   - Is not feasible in all cases.

Additionally, the inability to withdraw funds might be intentional but is uncommon and not explicitly stated in the contract documentation.

Impact

High.
The game cannot be properly started or played if the `TwentyOne` contract has insufficient funds.

Tools Used

- Manual Review
- Foundry
- ChatGPT

Recommendations

1. **Implement a Payable Constructor:**
   Allow the deployer to set the initial contract balance upon deployment. Example:

```
constructor() payable {
    // Initialize the contract with an initial balance
}
```

2. **Add receive() and fallback() Functions:** Enable the contract to receive funds directly and allow users to provide tips. Example:

```
receive() external payable {
    // Handle direct Ether transfers
}

fallback() external payable {
    // Handle fallback scenarios
}
```

3. **Implement a Withdrawal Mechanism:** Allow the contract owner to withdraw forfeited funds while ensuring the contract retains sufficient balance for gameplay. Use OpenZeppelin's Ownable and onlyOwner for this. Example:

```
function withdraw(uint256 amount) external onlyOwner {
 require(address(this).balance >= amount + MINIMUM_BALANCE, "
    Insufficient balance to withdraw");
 payable(owner()).transfer(amount);
 }
```

4. **Expose contract balance:** `solidity function getContractBalance()external view returns (uint256){ return address(this).balance; }`

5. **Update Documentation:** Clearly document the intended methods for funding and withdrawing, and highlight any limitations or intentional design choices. ——————

[H-2] TwentyOne::playersHand Calculation is Off by One for Cards from 2 Through 10 Inclusive

Summary

The `TwentyOne::playersHand` calculation incorrectly adds the `cardValue` for cards equivalent to the real-world values 2, 3, 4, 5, 6, 7, 8, 9, and 10. This issue arises because the `cardValue` calculation performs a modulo 13 operation on numbers 1 through 52, which are intended to represent a full deck

of cards, and directly adds the result to the `TwentyOne::playersHand::playerTotal`. The problem is that real-world cards do not have a value of 1, yet the modulo operation results in the value 1. This leads to the returned `TwentyOne::playersHand::playerTotal` being off by one for each drawn card with a value between 2 and 10 inclusive.

Vulnerability Details

The `TwentyOne::playersHand` calculation incorrectly adds the following `cardValue`:

```
uint256 cardValue = playersDeck[player].playersCards[i] % 13;
```

to the `TwentyOne::playersHand::playerTotal` for cards with values between 2 and 10. This is caused by the **`else`** condition in the following statement:

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

Issue

- Card values in the range [1, 9] (representing real-world cards 2 through 10) are incorrectly added to `playerTotal` as `cardValue`.
- The correct implementation should add these card values as `cardValue + 1`.
- Card values in the range [10, 12] should be added as `10`, representing the equivalent of the real-world cards Jack (J), Queen (Q), and King (K).
- Aces (value 0 in modulo 13) should be handled separately according to BlackJack rules. Refer to the H-3 finding for more details.

Correct Behavior

- For card values 1 through 9: `playerTotal += cardValue + 1;`
- For card values 10 through 12: `playerTotal += 10;`
- Aces (value 0): Handle separately based on `H-3`.

Impact

High.

This bug occurs 100% of the time and causes incorrect calculations for `TwentyOne::playersHand`.

This can result in: 1. Players losing when they should have won. 2. Players winning when they should have lost.

This undermines the fairness and reliability of the game.

Tools Used

1. Manual Review
2. Foundry (Proof-of-Concept unit tests can be provided upon request)
3. ChatGPT

Recommendations

1. **Rewrite the `if-else` Statements:**
   Update the `if-else` logic to properly calculate the `TwentyOne::playersHand::playerTotal` in line with BlackJack rules. The correct implementation should follow these rules:

   - Card values in the range [1, 9] (representing real-world cards 2 through 10) should be added as `cardValue + 1`.
   - Card values in the range [10, 12] (representing J, Q, K) should be added as `10`.
   - Aces (value 0) should be handled separately. Refer to my [H-3] finding

   Example correction:

   ```
   if (cardValue == 0) {
       // Handle Ace separately (refer to H-3)
   } else if (cardValue >= 10) {
       playerTotal += 10; // J, Q, K
   } else {
       playerTotal += cardValue + 1; // Cards 2 through 10
   ```

2. **Refer to my H-3 finding:**
   Ensure Ace handling is addressed as specified in the linked finding.

3. **Test Thoroughly:**
   Create unit tests using Foundry to verify:

   - Correct calculations for all card ranges.
   - Proper handling of Aces according to the my [H-3] finding.

[H-3] TwentyOne::playersHand Calculation Does Not Follow Official BlackJack Rules for Ace Calculation

Summary

In BlackJack, the Ace can have a value of either 11 or 1, depending on the current hand value of the player. If the current player's hand value plus 11 is less than or equal to 21, then the Ace should be counted as 11; otherwise, it is counted as 1. This rule is not honored in the TwentyOne::playersHand function, which adds the Ace as 10 in all cases.

Vulnerability Details

In the TwentyOne::playersHand function, the current card value is calculated as:

```
1  uint256 cardValue = playersDeck[player].playersCards[i] % 13
```

where the card value could be between 0 and 12. When the card value is 0 (representing the Ace as it is the 13th card), the following code adds 10 to the player's total:

```
1      if (cardValue == 0 || cardValue >= 10) {
2          playerTotal += 10;
3      } else {
4          playerTotal += cardValue;
5      }
```

This does not follow BlackJack rules, as the Ace should be added as either 11 or 1, depending on the player's current hand total.

Impact

High Impact: This bug occurs 100% of the time. The TwentyOne::playersHand calculation is incorrect, which may result in players losing when they should have won, or winning when they should have lost.

Tools Used

1. Manual Review

2. Foundry (PoC unit tests can be provided upon request)
3. ChatGPT

Recommendations

1. Update the logic to properly handle the cardValue for an Ace. The revised function may look as follows:

```
function playersHand(address player) public view returns (uint256) {
    uint256 playerTotal = 0;
    uint256 numberOfAces = 0;

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

        if (cardValue == 0) {
            // Ace
            numberOfAces += 1;
            playerTotal += 11;
        } else if (cardValue >= 10) {
            // Face cards: Jack, Queen, King
            playerTotal += 10;
        } else {
            playerTotal += cardValue + 1; // Offset 1 to map 1->2,
                2->3, ..., 9->10
        }
    }

    // Adjust for Aces if total exceeds 21
    while (playerTotal > 21 && numberOfAces > 0) {
        playerTotal -= 10;
        numberOfAces -= 1;
    }

    return playerTotal;
}
```

[H-4] TwentyOne::dealersHand calculcation is off by one for cards from 2 through 10 inclusive

Summary

The TwentyOne::dealersHand calculation wrongly adds the cardValue for the equivalent of the real world cards 2,3,4,5,6,7,8,9,10 because, the cardValue to add calculation, does modulo 13 division, of the numbers 1 through 52, which are supposed to represent a full card deck, and adds the result to the TwentyOne::dealersHand::dealerTotal. This is problematic because it does not account for real world cards not having a 1 value, while the modulo 13 division produces the result 1. This effectively results in returned TwentyOne::dealersHand::dealerTotal being one less than it should be, for each drawn card that is between 2,3,4,5,6,7,8,9,10.

Vulnerability Details

The TwentyOne::dealersHand calculation wrongly adds the following cardValue:

```
1      uint256 cardValue = dealersDeck[player].dealersCards[i] % 13;
```

to the TwentyOne::dealersHand::dealerTotal in the cases where the cardValue ends up between 1 and 10, due to “else” in this statement:

```
1      if (cardValue >= 10) {
2          dealerTotal += 10;
3      } else {
4          dealerTotal += cardValue;
5      }
```

Issue

- Card values in the range [1, 9] (representing real-world cards 2 through 10) are incorrectly added to `dealerTotal` as `cardValue`.
- The correct implementation should add these card values as `cardValue + 1`.
- Card values in the range [10, 12] should be added as `10`, representing the equivalent of the real-world cards Jack (J), Queen (Q), and King (K).
- Aces (value 0 in modulo 13) should be handled separately according to BlackJack rules. Refer to my H-5 finding below for more details.

Correct Behavior

- For card values 1 through 9: `dealerTotal += cardValue + 1;`
- For card values 10 through 12: `dealerTotal += 10;`
- Aces (value 0): Handle separately based on `[H-5]` finding.

Impact

This bug occurs 100% of the time and causes incorrect calculations for `TwentyOne::dealersHand`. This can result in:

1. Players losing when they should have won.
2. Players winning when they should have lost.

This undermines the fairness and reliability of the game.

Tools Used

1. Manual Review
2. Foundry (PoC unit tests can be provided upon request)
3. ChatGPT

Recommendations

1. **Rewrite the `if-else` Statements:**
   Update the `if-else` logic to properly calculate the `TwentyOne::dealersHand::dealerTotal` in line with BlackJack rules. The correct implementation should follow these rules:

   - Card values in the range [1, 9] (representing real-world cards 2 through 10) should be added as `cardValue + 1`.
   - Card values in the range [10, 12] (representing J, Q, K) should be added as `10`.
   - Aces (value 0) should be handled separately. Refer to my H-5 finding

---

[H-5] TwentyOne::dealersHand Calculation Does Not Follow Official BlackJack Rules for Ace Calculation

Summary

In BlackJack, the Ace can have a value of either 11 or 1, depending on the current hand value of the player. If the current player's hand value plus 11 is less than or equal to 21, then the Ace should be counted as 11; otherwise, it is counted as 1. This rule is not honored in the TwentyOne::dealersHand function, which adds the Ace as 10 in all cases.

Vulnerability Details

Inside TwentyOne::dealersHand the current card value is calculated with:

```
uint256 cardValue = dealersDeck[player].dealersCards[i] % 13;
```

where the card value could be between 0 and 12. When the card value is 0 (representing the Ace as it is the 13th card), the following code adds 10 to the dealer's total:

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

This does not follow BlackJack rules, as the Ace should be added as either 11 or 1, depending on the dealer's current hand total.

Impact

High Impact: This bug occurs 100% of the time. The TwentyOne::playersHand calculation is incorrect, which may result in players losing when they should have won, or winning when they should have lost.

Tools Used

1. Manual Review
2. Foundry (PoC unit tests can be provided upon request)
3. ChatGPT

Recommendations

1. Update the logic to properly handle the cardValue for an Ace.

Medium

[M-1] TwentyOne does not adhere to official BlackJack rules for a tie (called “push”)

Summary

TwentyOne::call function improperly deals with the case when dealerHand == playerHand. In such cases, according to official Blackjack rules, a tie (called a “push”) means the player neither wins nor loses their bet; their stake should be returned. Therefore, the code should be modified to handle ties by returning the player’s bet, rather than treating it as a loss.

Vulnerability Details

If a tie occurs it will end up in the final else of this block of code:

```
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);
}
```

which would make the player lose, even though they should get their funds back in such cases. The contract currently also does not support a way to return the players’ initial bet of 1 eth, so this needs to be implemented as well.

Impact

Medium as the likelyhood of a tie happening is not super high, however its crucial to be fixed as players wont be happy if they should get their money back, but they actually lose them.

Tools Used

1. Manual Review
2. Foundry (PoC unit tests can be provided upon request)
3. ChatGPT

Recommendations

1. The code should be modified to handle the tie (push) scenario correctly by returning the player's bet without any gain or loss. Correct logic should be something like this:

```
if (dealerHand > 21) {
    // Dealer busts, player wins
} else if (playerHand > dealerHand) {
    // Player's hand is higher, player wins
} else if (playerHand == dealerHand) {
    // Tie game, player's bet is returned
    // Implement logic to return the player's bet
} else {
    // Dealer's hand is higher, player loses
}
```

Low

[L-1] Solidity compiler version not static and other informational/gas improvements

Summary

Solidity compiler version is not explicity set to be unchangeable, which in extremy rare scenarios could lead to some unexpected behaviour or might lead to missing out on improvement possiblities. I will use this section to suggest other best practices that can be employed. While some of them might be out of scope of the security review they provide value by teaching the protocol creators about security and general best practices thus making web3 more secure.

Vulnerability Details

Caret (^) Operator: The caret symbol in ^0.8.13 allows the compiler to use any version from 0.8.13 up to (but not including) 0.9.0. This means the contract code could be compiled with newer compiler versions that you haven't tested with.

Risk of Breaking Changes: While minor versions (e.g., 0.8.x) are supposed to be backward compatible, there can still be subtle changes or deprecations that affect the contract's behavior.

Impact

Minimal/Extremely unlikely to cause any issues

Tools Used

1. Manual review
2. Foundry toolkit (anvil, forge test, etc)
3. ChatGPT

Recommendations

1. Follow best practice of Using Exact Solidity Compiler Version: Replace the caret pragma with a specific version. For example: pragma solidity 0.8.13;
2. Follow best practice of using custom errors instead of require. This can save Gas, while being as verbose, for example

```
    require(
        availableCards[player].length == 0,
        "Player's deck is already initialized"
    );
```

could be re-written to something like:

```
error TwentyOne__PlayerDeckInitilized();
....
    if (availableCards[player].length == 0) {
        revert TwentyOne__PlayerDeckInitilized();
    }
```

3. Test suite is not passing. Just donwloading the code and running forge test has one test (test_Call_PlayerWins) failling, due to inssuficient funds in the TwentyOne contract. More info on

this can be found in my H-1 report. Having good test suite that includes unit & integration tests as well as fuzz tests and some sort of formal verification is crucial for ensuring project security. This can be fixed by:

4. Deploy scripts not leveraged for testing. Deploy script can be very helpful for having a streamlined DevOps process for the protocol.
5. Consider using private function visibility if the contract is not meant to be inherited
6. Consider using indexed params (player addresses might be a good idea) in events for better trackability of events
7. Consider using constants for repeated numbers and magic numbers throughout the code
8. Adding Explicit visibility for variables like playersDeck and dealersDeck is considered best practice
9. Consider using natspec

Informational

Check the Recommendations section for the Low finding # Gas Check the Recommendations section for the Low finding