Protocol Audit Report

Prepared by: OSTE (Oudjani Seyyid taqy eddine)

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. The game involves a player competing against a dealer, with standard blackjack rules applied. A random card drawing mechanism is implemented to mimic shuffling, and players can choose to "hit" or "stand" based on their card totals. Winning players double their wager, while losing players forfeit their initial bet.

Disclaimer

The OSTE 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
  TwentyOne.sol
```

Roles

Actors:

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

Executive Summary

Issues found

Severity	Number of issues found
High	2
Medium	1
Low	0
Info	2
Gas Optimizations	1
Total	6

Findings

High

[H-1] The use of Poor method for generating Random number `TwentyOne::drawCard()` to draw a card can be manuplated and predicted by miner

Description:

1. Predictable Inputs

- block.timestamp: The block timestamp can be influenced to a small degree by miners (within ~15 seconds of variability), allowing them to manipulate the randomness outcome.
- msg.sender: The address of the caller is deterministic and fully controlled by the user executing the transaction.
- block.prevrandao: Although this is sourced from the randomness beacon introduced in Ethereum's PoS consensus, it may still be insufficiently random when combined with the other weak factors.

2. Miner Manipulation

   Miners have control over both block.timestamp and block.prevrandao to some extent. By strategically adjusting these values, they can bias the randomness to favor a particular outcome. For example, if an attacker is also the miner, they can influence the random index to draw specific cards for the player or the dealer.

```
        uint256 randomIndex = uint256(
@>            keccak256(
                abi.encodePacked(block.timestamp, msg.sender,
```

```
  block.prevrandao)
              )
          ) % availableCards[player].length;


```

Impact:

- Drawing a random card for either the dealer or player can be manipulated by an attacker to obtain a desired card, enabling the player to win.
- An attacker can exploit this weakness to predict or manipulate the outcome of drawCard(). For example, they can ensure the player receives high-value cards while the dealer gets low-value cards, thereby guaranteeing a win for the attacker.

Proof of Concept:

Recommended Mitigation:

1. Use Chainlink VRF: Integrate Chainlink's Verifiable Random Function (VRF) to generate cryptographically secure random numbers that are unpredictable and tamper-proof.

```

  uint256 randomIndex = VRFConsumer.getRandomNumber() %
  availableCards[player].length;

```

2. Avoid On-Chain Only Randomness: Avoid relying solely on on-chain inputs like block.timestamp and block.prevrandao. Combine with off-chain randomness for stronger entropy.

[H-2] Player Can lost ETH if start the game with more than 1 eth as Value `TwentyOne::startGame()`

Description: Although the game logic specifies that a player can only start with exactly 1 ETH and win double that amount if victorious, the contract currently allows players to send more than 1 ETH when calling startGame(). Any excess ETH sent is not accounted for or refunded, resulting in the player losing the additional funds, even if they win.

```

    function startGame() public payable returns (uint256) {
          _;
@>          require(msg.value >= 1 ether, "not enough ether sent");
          _;
      }

```

Impact:

1. Players who inadvertently send more than 1 ETH will lose the excess amount, leading to financial losses.
2. This could harm the contract's reputation, reducing player trust and deterring users from engaging with the game.
3. ewer players might participate, negatively affecting the adoption and success of the game.

Proof of Concept:

1. Add this POC to TwentyOne.t.sol

```
function test_Call_PlayerWins() public {
        vm.startPrank(player1); // Start acting as player1

        twentyOne.startGame{value: 5 ether}();

        // Mock the dealer's behavior to ensure player wins
        // Simulate dealer cards by manipulating state
        vm.mockCall(
            address(twentyOne),
            abi.encodeWithSignature("dealersHand(address)", player1),
            abi.encode(18) // Dealer's hand total is 18
        );

        uint256 initialPlayerBalance = player1.balance;

        // Player calls to compare hands
        twentyOne.call();

        // Check if the player's balance increased (prize payout)
        uint256 finalPlayerBalance = player1.balance;
        assertGt(finalPlayerBalance, initialPlayerBalance);
        console.log(finalPlayerBalance, initialPlayerBalance);
        vm.stopPrank();
    }
```

2. Run the command :

```
Forge test
```

3. Notice the player entered with 5 ETH and although he win he get only 2 ETH as balance and instead it should be 6.

Recommended Mitigation:

1. Restrict the game start to exactly 1 ETH and reject transactions with any other amount.

```
require(msg.value == 1 ether, "You must send exactly 1 ETH to start the
game");
```

2. Alternatively, refund any excess ETH to the sender:

```
uint256 excess = msg.value - 1 ether;
if (excess > 0) {
    payable(msg.sender).transfer(excess);
}
```

Medium

[M-1] The use of Poor method for generating Random number `TwentyOne::call()` to Select a Trashold for the dealer can be manuplated and predicted by miner

Description:

1. Predictable Inputs

- block.timestamp: The block timestamp can be influenced to a small degree by miners (within ~15 seconds of variability), allowing them to manipulate the randomness outcome.
- msg.sender: The address of the caller is deterministic and fully controlled by the user executing the transaction.
- block.prevrandao: Although this is sourced from the randomness beacon introduced in Ethereum's PoS consensus, it may still be insufficiently random when combined with the other weak factors.

2. Miner Manipulation

   Miners have control over both block.timestamp and block.prevrandao to some extent. By strategically adjusting these values, they can bias the randomness to favor a particular outcome.

```
  uint256 standThreshold = (uint256(
@>            keccak256(
```

```
                abi.encodePacked(block.timestamp, msg.sender,
block.prevrandao)
            )
        ) % 5) + 17;
```

Impact:

An attacker can manipulate the contract to select a threshold value in their favor based on the cards they hold. By exploiting the predictable random number generation or controllable variables within the contract, the attacker can influence the outcome of the threshold determination process. This allows them to adjust the game logic or decision-making mechanisms in a way that maximizes their chances of winning or minimizes their losses.

Such manipulation undermines the fairness of the game and creates an unfair advantage, potentially enabling consistent exploitation to drain funds, disrupt gameplay, or deceive other participants. It highlights a critical issue in the reliance on weak or manipulable randomness within smart contracts.

Proof of Concept:

Recommended Mitigation:

1. Use Chainlink VRF: Integrate Chainlink's Verifiable Random Function (VRF) to generate cryptographically secure random numbers that are unpredictable and tamper-proof.

```
uint256 randomIndex = VRFConsumer.getRandomNumber()  % 5 + 17;
```

2. Avoid On-Chain Only Randomness: Avoid relying solely on on-chain inputs like block.timestamp and block.prevrandao. Combine with off-chain randomness for stronger entropy.

Informational

[i-1] Using floating Pragma is considered bad practices

Description: Using an unfixed Solidity version, such as pragma solidity ^0.8.13;, allows the contract to compile with any minor version of the compiler from 0.8.13 up to (but not including) 0.9.0. While this offers flexibility, it introduces potential risks due to changes or bugs introduced in future minor versions of Solidity.

Newer compiler versions may:

```
Introduce unexpected breaking changes in behavior or optimizations.
Contain undiscovered vulnerabilities or newly introduced bugs that could
```

```
impact the contract's functionality.
Alter how the contract interacts with other components or external
contracts, leading to inconsistencies.
```

Impact:

1. Security Risks:

- Contracts may become vulnerable to bugs or security issues introduced in newer Solidity versions after deployment.
- Incompatibility with external libraries or dependencies if they rely on specific compiler behaviors.

2. Operational Risks:

- Future Solidity updates could unintentionally alter the behavior of the contract during re-deployment or interactions in testing environments.

3. Audit and Maintenance Challenges:

- Auditors and developers may face uncertainty about the exact compiler version used to deploy the contract, complicating bug reproduction or code validation.

Recommended Mitigation:

Pin Specific Compiler Version: Use an exact version of Solidity in the pragma statement to ensure consistent behavior and reproducibility.

```
pragma solidity 0.8.13;
```

[i-2] Using Magic Number instead of const

Description: A magic number is a hardcoded numeric value used directly in the code without explanation. Magic numbers reduce the readability and maintainability of the code because they lack descriptive context. If the value needs to be changed or reused, developers must hunt through the code to find all instances of the number, increasing the risk of errors.

```
    function playersHand(address player) public view returns (uint256) {
            _;
            if (cardValue == 0 || cardValue >= 10) {
@>              playerTotal += 10;
            }
            _;
```

```
    function dealersHand(address player) public view returns (uint256) {
            _;
            if (cardValue == 0 || cardValue >= 10) {
@>              playerTotal += 10;
            }
            _;
```

Impact:

1. Reduced Code Readability: Other developers or auditors may struggle to understand the purpose of the hardcoded value, making the code harder to interpret and debug.
2. `Increased Risk of Errors:`

If the same magic number is used in multiple places and needs to be updated, there's a higher chance of missing one instance or inconsistently applying the update.

3. Lack of Flexibility: Magic numbers make it difficult to change behavior dynamically or adapt to new requirements since the hardcoded values aren't centralized.

Recommended Mitigation:

1. use a const with values of all magic numbers.

Gas

[Gas-3] Using Loop in `TwentyOne::initializeDeck()` each time the game start can cause higher cos of gas

Description: The initializeDeck() function in the TwentyOne contract uses a loop to populate a player's deck with all cards from a standard deck (1 to 52). This approach incurs significant gas costs because:

1. Storage writes (e.g., availableCards[player].push(i)) are among the most expensive operations in Solidity.
2. Each iteration of the loop performs a write operation, which is executed 52 times, leading to excessive gas consumption.

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

Impact:

1. Increased Gas Costs for Users: Players or game operators will pay significantly more gas every time the deck is initialized, reducing user satisfaction and scalability.

Proof of Concept:

Recommended Mitigation:

Consider using immutable deck variable instead of using list each initializations :

```
uint256[52] public constant DECK = [
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
    14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
    27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
    40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52
];
availableCards[player] = DECK
```