Protocol Audit Report

Version 1.0

MrSaade

Protocol Audit Report

MrSaade

September 21, 2024

Prepared by: [MrSaade]

Table of Contents

* [I-1] Dead Code in _isActivePlayer Function
* [I-2] Magic Numbers Used in selectWinner Function
* [I-3] Cache Array Length in Loops to Save Gas
* [I-4] Old and floating Solidity Version Used (^0.7.6)
* [I-5] Declare State Variables as constant or immutable

Protocol Summary

PuppyRaffle is a decentralized smart contract-based raffle system where users can enter to win a unique NFT puppy. Participants pay an entry fee to join the raffle, and after a set duration, a random winner is selected to receive the NFT. The contract allows for multiple participants, ensures fairness by restricting duplicate entries, and enables refunds. The protocol also includes a fee mechanism where a portion of the entry fees is collected and distributed to a designated fee address.

Disclaimer

This audit report is intended to provide information about potential security vulnerabilities in the PuppyRaffle smart contract. It is not an exhaustive review, and no guarantee is provided regarding the absence of undiscovered vulnerabilities. The responsibility for fixing issues and securing the contract lies solely with the project team. Use of this report or its recommendations is at the discretion of the project team, and the auditor assumes no liability for any consequences arising from the use of this report.

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

The findings described in this document correspond to the following commit hash:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

```
./src/
-- PuppyRaffle.sol
```

Roles

- Owner: The only one who can change the `feeAddress`, denominated by the `_owner` variable.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

Executive Summary

This audit of the PuppyRaffle smart contract revealed several critical, medium, and low-severity vulnerabilities, along with gas optimization opportunities. Key issues include reentrancy vulnerabilities in the refund process, insecure randomness in winner selection, potential for Denial of Service (DoS) attacks due to gas exhaustion, and improper handling of certain edge cases, such as smart contract winners without fallback functions. Additionally, the audit identified potential improvements in gas efficiency.

Addressing these vulnerabilities will significantly enhance the security and performance of the protocol, ensuring a fairer, more robust, and cost-efficient raffle system.

Issues found

Severity	Number of issues found
High	4
Medium	1

Severity	Number of issues found
Low	0
Info	5
Total	10

Findings

[H-1] Reentrancy in `PuppyRaffle.sol::refund` Function can drain contract funds

Description The refund function allows users to claim refunds, but state changes occur after sending the ether. This exposes the contract to reentrancy attacks where an attacker can repeatedly call the refund function before the state is updated, allowing them to repeatedly claim refunds and drain the contract's balance.

```
1  function refund(uint256 playerIndex) public {
2        address playerAddress = players[playerIndex];
3        require(
4            playerAddress == msg.sender,
5            "PuppyRaffle: Only the player can refund"
6        );
7        require(
8            playerAddress != address(0),
9            "PuppyRaffle: Player already refunded, or is not active"
10       );
11    @>   payable(msg.sender).sendValue(entranceFee); //Reenetrancy
          vulnearability
12
13    @>   players[playerIndex] = address(0); // State updated after
          external call
14       emit RaffleRefunded(playerAddress);
15    }
```

Impact An attacker can reenter the function before the state is updated, allowing them to repeatedly claim refunds and drain the contract's balance of all fees.

Proof of Concepts Added the following test code to the `PuppyRaffleTest.t.sol` file

Test Code

```
1  contract ReentrancyAttacker {
2      PuppyRaffle puppyRaffle;
3      uint256 entranceFee;
4      uint256 attackerIndex;
```

```
5
6       constructor(address _puppyRaffle) {
7           puppyRaffle = PuppyRaffle(_puppyRaffle);
8           entranceFee = puppyRaffle.entranceFee();
9       }
10
11      function attack() external {
12          address;
13          players[0] = address(this);
14          puppyRaffle.enterRaffle{value: entranceFee}(players);
15          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
16          puppyRaffle.refund(attackerIndex);
17      }
18
19      receive() external payable {
20          if (address(puppyRaffle).balance >= entranceFee) {
21              puppyRaffle.refund(attackerIndex);
22          }
23      }
24  }
25  function testReentrance() public playersEntered {
26      ReentrancyAttacker attacker = new ReentrancyAttacker(address(
            puppyRaffle));
27      vm.deal(address(attacker), 1e18);
28      uint256 startingAttackerBalance = address(attacker).balance;
29      uint256 startingContractBalance = address(puppyRaffle).balance;
30      console.log("Starting attacker balance: ", startingAttackerBalance)
            ;
31      console.log("Starting contract balance: ", startingContractBalance)
            ;
32      attacker.attack();
33      uint256 endingAttackerBalance = address(attacker).balance;
34      uint256 endingContractBalance = address(puppyRaffle).balance;
35      console.log("Ending attacker balance: ", endingAttackerBalance);
36      console.log("Ending contract balance: ", endingContractBalance);
37      assertEq(endingAttackerBalance, startingAttackerBalance +
            startingContractBalance);
38      assertEq(endingContractBalance, 0);
39  }
```

Recommended mitigation Apply the checks-effects-interactions pattern by updating the state before sending the ether. Thus, the state must be updated before the external call `payable(msg.sender).sendValue(entranceFee);` is made.

```
1   function refund(uint256 playerIndex) public {
2           address playerAddress = players[playerIndex];
3           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund");
4           require(playerAddress != address(0), "PuppyRaffle: Player
```

```
                already refunded, or is not active");
+           players[playerIndex] = address(0);
+           emit RaffleRefunded(playerAddress);
            payable(msg.sender).sendValue(entranceFee);
-           players[playerIndex] = address(0);
-           emit RaffleRefunded(playerAddress);
}
```

[H-2] Insecure/Weak Randomness in `PuppyRaffle.sol::selectWinner` function allows manipulation

Description The `PuppyRaffle.sol::selectWinner` function uses keccak256 combined with on-chain data (msg.sender, block.timestamp, and block.difficulty). These on-chain data are predictable and manipulatable by validators, leading to weak randomness.

Impact Attackers/miners/validators can manipulate the outcome of the raffle by predicting the winner as well as manipulating the type of puppy Nft to be minted. This undermines the integrity of the raffle system as really rare nfts are more likely to be selected and true winners are less likely to be selected, making it unfair for other participants who do not have the same level of control. The fairness and unpredictability of the raffle system are critical, and the current randomness mechanism is highly susceptible to manipulation.

Proof of Concept Predictability of block.timestamp and block.difficulty: Validators/Attackers can predict or manipulate these values. Validators can know the value of block.timestamp and block.difficulty in advance because they have control over when the block is mined. Since block.difficulty has been replaced by prevrandao, it still carries the same vulnerability, where miners or validators can influence the outcome by controlling the timing and context of block creation. This predictability allows them to calculate the result of the randomness function ahead of time and only participate when it is in their favor.

Manipulation of msg.sender: Users can also manipulate the `msg.sender` value to increase the likelihood of winning the raffle. Since the `msg.sender` value directly influences the outcome of the winner selection, a malicious participant could call the `PuppyRaffle.sol::selectWinner`function multiple times using different accounts or contract addresses, effectively increasing their chances of becoming the selected winner.

Recommended mitigation To address this, the contract should adopt a secure and verifiable source of randomness, such as Chainlink VRF (Verifiable Random Function), which provides randomness that is both unpredictable and tamper-proof. Using commit-reveal schemes can also be a viable alternative to mitigate these issues by decoupling the selection process from immediate on-chain data.

[H-3] Unsafe Casting of uint256 to uint64 and Integer Overflow Risk in `PuppyRaffle.sol::selectWinner` function could lead to loss of funds

Description The `PuppyRaffle::selectWinner` function includes a dangerous cast from uint256 to uint64 when updating the `PuppyRaffle::totalFees` variable. This can lead to an integer overflow when the total fees exceed the maximum value that a uint64 can store (18,446,744,073,709,551,615). Additionally, Ethereum operates with 18 decimal places, so transactions involving significant amounts of ether could cause an overflow. This could result in incorrect fee tracking and loss of funds.

Impact If the `PuppyRaffle::totalFees` exceed the maximum value for a uint64, this will cause an overflow, leading to loss of track of fees. This could result in underpayment or overpayment of fees, and potentially lock funds in the contract.

Proof of Concepts Added this test code to the `PuppyRaffleTest.t.sol` file:

Test Code

```
function testTotalFeesOverflow() public playersEntered {
    // We finish a raffle of 4 to collect some fees
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
    puppyRaffle.selectWinner();
    uint256 startingTotalFees = puppyRaffle.totalFees();
    // startingTotalFees = 800000000000000000 wei / 0.8 ether(20 % 4 ether)

    // We then have 89 players enter a new raffle
    uint256 playersNum = 89;
    address[] memory players = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        players[i] = address(i);
    }
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
    puppyRaffle.selectWinner();

    uint256 endingTotalFees = puppyRaffle.totalFees();
    assert(endingTotalFees < startingTotalFees); //endingTotalFees will be significantly less than startingTotalFees due to overflow

    // We are also unable to withdraw any fees because of the require check

    vm.expectRevert("PuppyRaffle: There are currently players active!");
    puppyRaffle.withdrawFees();

}
```

Recommended mitigation -Upgrade to Solidity 0.8.x, which has built-in overflow/underflow checks to prevent these types of errors.

```
-pragma solidity ^0.7.6;
+pragma solidity 0.8.20;
```

-Avoid unsafe casting of large values like uint256 to uint64. Use uint256 consistently to avoid overflow issues and remove the casting altogether.

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;

- totalFees = totalFees + uint64(fee);
+ totalFees = totalFees + fee;
```

-Remove the balance check in `PuppyRaffle::withdrawFees`

```
- require(address(this).balance == uint256(totalFees), "PuppyRaffle:
    There are currently players active!");
```

[H-4] Malicious Winner Can Halt the Raffle

Description: This high-severity finding highlights two ways in which a malicious participant can halt the PuppyRaffle by exploiting how the contract interacts with smart contracts when selecting a winner.

1.External Call to Winner: When the `PuppyRaffle::selectWinner` function chooses a winner, it attempts to transfer the prize money to the winner using an external call:

```
@> (bool success,) = winner.call{value: prizePool}("");
require(success, "PuppyRaffle: Failed to send prize pool to winner");
```

If the winner is a smart contract that does not implement a payable receive function or fallback function, or if these functions are intentionally designed to revert the call, the external transfer will fail. This will trigger the require statement and revert the entire selectWinner transaction, preventing the funds from being sent.

Impact: The prize money will never be transferred to the winner, and as a result, the raffle will not reset to allow a new round to begin. This effectively halts the entire raffle, as the function cannot complete without transferring the prize.

2.Minting an NFT to the Winner: In addition to transferring funds, the selectWinner function also mints an NFT to the winner using the _safeMint function::

```
@> _safeMint(winner, tokenId);
```

The _safeMint function is part of the ERC721 standard and checks if the winner's address is a smart contract. If the winner is a smart contract, it will attempt to call the onERC721Received hook on the contract.

If the smart contract does not implement the onERC721Received function, or if it deliberately causes this function to revert, the minting of the NFT will fail, causing the entire selectWinner function to revert.

Impact: As with the prize transfer issue, failing to mint the NFT will prevent the raffle from completing, and a new round will not begin.

Impact Summary: In both scenarios (whether the prize transfer or NFT minting fails), the entire raffle can be halted permanently: Scenario 1: The winner is a smart contract that does not accept ether transfers. This causes the prize to never be distributed, and the raffle never resets for a new round. Scenario 2: The winner is a smart contract that does not implement the onERC721Received hook, causing the minting of the NFT to fail and halting the raffle indefinitely.

Proof of Concepts: This test code will pass without the duplicate check in the `PuppyRaffle::enterRaffle` function:

Test Code

```
function testSelectWinnerDoS() public {
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    address[] memory players = new address[](4);
    players[0] = address(new AttackerContract());
    players[1] = address(new AttackerContract());
    players[2] = address(new AttackerContract());
    players[3] = address(new AttackerContract());
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

    vm.expectRevert();
    puppyRaffle.selectWinner();
}

contract AttackerContract {
    // Implements a `receive` function that always reverts
    receive() external payable {
        revert();
    }
}

//Or this
contract AttackerContract {
    // Implements a `receive` function to receive prize, but does not
       implement `onERC721Received` hook to receive the NFT.
    receive() external payable {}
```

```
27 }
```

Recommended Mitigation: To prevent this attack, the following mitigations should be considered:

Use the Pull Over Push Pattern: Instead of sending the prize directly to the winner via an external call, allow winners to manually claim their prize by calling a withdraw function. This would avoid any potential issues with ether transfers to incompatible or malicious smart contracts.

```
1  // Mapping to store the prize pool for each winner
2  + mapping(address => uint256) public winnings;
3
4  // selectWinner function modified to use the winnings mapping
5  function selectWinner() external {
6      require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
7      require(players.length >= 4, "PuppyRaffle: Need at least 4 players"
           );
8      uint256 winnerIndex =
9          uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
               block.difficulty))) % players.length;
10     address winner = players[winnerIndex];
11     uint256 totalAmountCollected = players.length * entranceFee;
12
13     uint256 prizePool = (totalAmountCollected * 80) / 100;
14     uint256 fee = (totalAmountCollected * 20) / 100;
15     totalFees = totalFees + uint64(fee);
16     // Store the prize in the winnings mapping for the winner
17 +    winnings[winner] += prizePool;
18
19     uint256 tokenId = totalSupply();
20     uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
           block.difficulty))) % 100;
21     if (rarity <= COMMON_RARITY) {
22         tokenIdToRarity[tokenId] = COMMON_RARITY;
23     } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
24         tokenIdToRarity[tokenId] = RARE_RARITY;
25     } else {
26         tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
27     }
28
29     delete players;
30     raffleStartTime = block.timestamp;
31     previousWinner = winner;
32     (bool success,) = winner.call{value: prizePool}("");
33     require(success, "PuppyRaffle: Failed to send prize pool to winner"
           );
34     _safeMint(winner, tokenId);
35 }
36
37 // Function for the winner to claim their prize
38 +function claimPrize() public {
```

```
39 +    uint256 prize = winnings[msg.sender];
40 +   require(prize > 0, "No prize available to claim");
41 +
42 +   // Reset the prize to zero before transferring to prevent
       reentrancy
43 +  winnings[msg.sender] = 0;
44 +
45 +   // Transfer the prize to the winner
46 +  (bool success, ) = msg.sender.call{value: prize}("");
47 + require(success, "Failed to claim prize");
48 +}
```

[M-1] Denial of Service (DoS) Risk in enterRaffle Due to Unbounded Array

Description: The `PuppyRaffle::enterRaffle` function loops through the `PuppyRaffle::newPlayers` array, allowing users to submit multiple addresses. If a large number of addresses are submitted or as the size of the array increases significantly, this loop could consume excessive gas making the function unusable to later entrants leading to a Denial of Service (DoS) attack

Additionally, the increased gas cost introduces front-running opportunities where malicious users can detect a transaction about to be confirmed and submit their own transaction with a higher gas fee. This causes the victim's transaction to fail or increase their costs significantly.

Impact: The `PuppyRaffle::enterRaffle` function may become uncallable for users if a malicious participant submits a very large array, leading to gas exhaustion. Malicious actors can front-run other entrants by submitting their transactions with higher gas fees, causing the victim's transaction to fail, leading to an unfair and costly race condition.

Proof of Concept Added the following test code to the `PuppyRaffleTest.t.sol` file

Test Code

```
1  function testDoSAttackWithManyPlayers() public {
2      // Start with a reasonable number of players, then gradually
           increase
3      uint256 numberOfPlayers = 100;
4      address[] memory players = new address[](numberOfPlayers);
5
6      // Assign dummy addresses for testing purposes
7      for (uint256 i = 0; i < numberOfPlayers; i++) {
8          players[i] = address(uint160(i + 1)); // Convert uint256 to a
               valid address
9      }
10
11     // Start measuring gas consumption as we enter more players
12     uint256 gasBefore = gasleft();
```

```
13      puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(
            players); // Add players to the raffle
14      uint256 gasAfter = gasleft();
15      uint256 FirstGasUsed = gasBefore - gasAfter;
16
17      emit log_named_uint("Gas used for entering players", FirstGasUsed);
18
19      // Repeat the process with more players to observe increasing gas
            consumption
20      numberOfPlayers = 200; // Increase the number of players for
            further testing
21      players = new address[](numberOfPlayers);
22      for (uint256 i = 0; i < numberOfPlayers; i++) {
23          players[i] = address(uint160(i + 101)); // Ensure new unique
                addresses
24      }
25
26      // Expect gas costs to increase significantly
27      gasBefore = gasleft();
28      puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(
            players);
29      gasAfter = gasleft();
30      uint256 LatterGasUsed = gasBefore - gasAfter;
31
32      emit log_named_uint(
33          "Gas used for entering more players",
34          LatterGasUsed
35      );
36      assertGt(LatterGasUsed, FirstGasUsed);
37  }
```

Recommended Mitigation Use a mapping to track participants and their entries rather than looping through an unbounded array. Additionally, implement a reasonable upper limit on the number of participants that can be submitted in one transaction. Alternatively, you could use OpenZeppelin's EnumerableSet library.

Informational / Non-Critical

[I-1] Dead Code in _isActivePlayer Function

Description: The `PuppyRaffle::_isActivePlayer` function is never used in the contract, which indicates dead code.

Impact: Leaving dead code in the contract increases gas costs and complicates readability.

Recommended Mitigation: Remove the `PuppyRaffle::_isActivePlayer` function as it serves no purpose.

```
-function _isActivePlayer() internal view returns (bool) {
-        for (uint256 i = 0; i < players.length; i++) {
-           if (players[i] == msg.sender) {
-                return true;
-            }
-        }
-        return false;
-}
```

[I-2] Magic Numbers Used in selectWinner Function

Description: The selectWinner function uses hardcoded values for determining the percentage distribution between the prize pool and fees.

Impact: Using magic numbers reduces readability and makes it difficult to update the logic in the future.

Recommended Mitigation: Declare constants for the fee percentage and prize pool percentage, making the contract easier to maintain and modify.

```
+       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+       uint256 public constant FEE_PERCENTAGE = 20;
+       uint256 public constant TOTAL_PERCENTAGE = 100;
.
.
.
-        uint256 prizePool = (totalAmountCollected * 80) / 100;
-        uint256 fee = (totalAmountCollected * 20) / 100;
         uint256 prizePool = (totalAmountCollected *
            PRIZE_POOL_PERCENTAGE) / TOTAL_PERCENTAGE;
         uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
            TOTAL_PERCENTAGE;
```

[I-3] Cache Array Length in Loops to Save Gas

Description: Multiple loops reference `PuppyRaffle::players.length` in each iteration, which reads from storage, increasing gas costs.

Impact: Repeated access to storage variables increases gas consumption, especially as the size of the array grows.

Recommended Mitigation: Cache the length of the array in a local variable before the loop to minimize gas costs.

[I-4] Old and floating Solidity Version Used (^0.7.6)

Description: The contract is written in Solidity 0.7.6, which lacks several optimizations and safety features of newer versions.

Impact: Using an old version of Solidity `^0.7.6` prevents the compiler from checking for potential overflows and underflows when performing arithmetic operations where in this case, caused the loss of track of fees resulting in underpayment or overpayment of fees, and potentially lock funds in the contract.

Recommended Mitigation: Consider upgrading to Solidity 0.8.x to take advantage of overflow/underflow protections and other security improvements.

```
- pragma solidity ^0.7.6;
+ pragma solidity 0.8.20;
```

[I-5] Declare State Variables as constant or immutable

Description: The variables `PuppyRaffle::commonImageUri`, `PuppyRaffle::rareImageUri`, `PuppyRaffle::legendaryImageUri`, and `PuppyRaffle::raffleDuration` could be declared as constant or immutable since they are never modified after deployment.

Impact: Marking these variables as constant or immutable reduces gas costs as they are stored directly in bytecode.