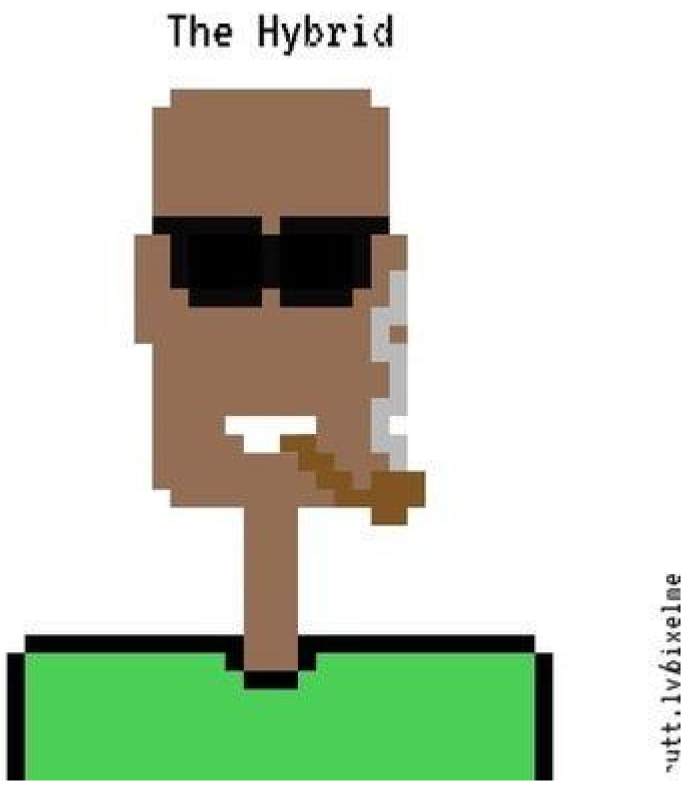

Protocol Audit Report

Prepared by: th3hybrid

Prepared by: th3hybrid

Lead Security Researcher:

- th3hybrid

Table of Contents

Protocol Summary

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

1. Call the `enterRaffle` function with the following parameters:
   1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The YOUR_NAME_HERE 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

		Impact		
	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:

```
2a47715b30cf11ca82db148704e67652ad679cd8
```

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	1
Gas	2
Info	7
Total	18

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance.

Description: The `PuppyRaffle::refund` function does not follow CEI (Checks,Effects,Interactions) 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 before updating `PuppyRaffle::players` 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 has entered the raffle could have a `fallback/receive` function that keeps calling `PuppyRaffle::refund` function until the contract balance is drained.

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

Proof of Concept:

1. User enters the raffle
2. Attacker sets up a contract with a `receive` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract,draining the contract balance.

Proof of Code:

▶ Code

Place the following into `PuppyRaffleTest.t.sol`

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

        uint256 startingAttackerBalance = address(attacker).balance;
        uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;
        vm.prank(attackUser);
        attacker.attack{value: entranceFee}();
        uint256 endingAttackerBalance = address(attacker).balance;
        uint256 endingPuppyRaffleBalance = address(puppyRaffle).balance;

        assertEq(endingAttackerBalance, entranceFee + startingPuppyRaffleBalance);
        console.log("startingAttackerBalance: ", startingAttackerBalance);
        console.log("startingPuppyRaffleBalance: ", startingPuppyRaffleBalance);
        console.log("endingAttackerBalance: ", endingAttackerBalance);
        console.log("endingPuppyRaffleBalance: ", endingPuppyRaffleBalance);
    }
```

And this contract too

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

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

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

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

Recommended Mitigation: To prevent 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.

```
    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);
+       emit RaffleRefunded(playerAddress);
        payable(msg.sender).sendValue(entranceFee);
-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

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

Description: Hashing the `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable random number. A predicatable random number is not a good random number, malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Note: This additionally means users could front-run this function and call `refund` if they see they are not the winner.

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffles.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See the solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.
2. Users can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner.
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Using on-chain values as randomness seed is a well-documented attack vector in the blockchain space.

Recommended Mitigation: Consider using a cryptographically provable random number generator such as 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;
//18446744073709551615
myVar = myVar + 1
//myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::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 conclude a raffle of 4 players
2. we then have 89 players enter a new raffle, and conclude the raffle
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 not be able to withdraw, due to the 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 to match and withdraw the fees, this is clearly not the intended design of the protocol. At some point, there will be too much `balance` in the contract that the above will be impossible to meet.

▶ Proof of Code

Place the following into `PuppyRaffleTest.t.sol`

```
    function test_canOverflowFees() public playersEntered {
        console.log("Fees: ", uint256(puppyRaffle.totalFees()));
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        address[] memory players = new address[](89);
        for (uint256 i = 0; i < 89; i++) {
            players[i] = address(uint160(i + 5));
        }
        puppyRaffle.enterRaffle{value: entranceFee * 89}(players);

        uint256 expectedFees = (93 * entranceFee * 20) / 100;

        puppyRaffle.selectWinner();
        uint256 actualFees = puppyRaffle.totalFees();
        console.log("Expected Fees: ", expectedFees);
        console.log("Actual Fees: ", actualFees);
        assert(actualFees < expectedFees);
        // 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.

```diff
- 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`.

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

3. Remove the balance check in `PuppyRaffle::withdrawFees`

```diff
- require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
```

There are more attack vectors with that final require, so we recommend removing it regardless.

[H-4] `PuppyRaffle::refund` replaces an index with address(0) which can cause the function `PuppyRaffle::selectWinner` to always revert

Description: `PuppyRaffle::refund` is supposed to refund a player and remove him from the current players. But instead, it replaces his index value with address(0) which is considered a valid value by solidity. This can cause a lot issues because the players array length is unchanged and address(0) is now considered a player.

```
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);
        }
```

If a player refunds his position, the function PuppyRaffle::selectWinner will always revert. Because more than likely the following call will not work because the prizePool is based on a amount calculated by considering that that no player has refunded his position and exit the lottery. And it will try to send more tokens that what the contract has :

```
uint256 totalAmountCollected = players.length * entranceFee;
uint256 prizePool = (totalAmountCollected * 80) / 100;

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

However, even if this calls passes for some reason (maybe there are more native tokens than what the players have sent or because of the 80% ...). The call will thankfully still fail because of the following line that doesn't allow NFT minting to the zero address.

```
 _safeMint(winner, tokenId);
```

Impact: The lottery is stopped, any call to the function `PuppyRaffle::selectWinner` will revert. There is no actual loss of funds for users as they can always refund and get their tokens back. However, the protocol is shut down and will lose all it's customers. A core functionality is broken, Impact is high.

Proof of Code:

▶ Code

Place the following into `PuppyRaffleTest.t.sol`

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

        // There are four winners. Winner is last slot
        vm.prank(playerFour);
        puppyRaffle.refund(3);

        // reverts because out of Funds
        vm.expectRevert();
        puppyRaffle.selectWinner();

        vm.deal(address(puppyRaffle), 10 ether);
        vm.expectRevert("ERC721: mint to the zero address");
        puppyRaffle.selectWinner();

    }
```

Recommended Mitigation: Delete the player index that has refunded.

```diff

-   players[playerIndex] = address(0);

+    players[playerIndex] = players[players.length - 1];
+    players.pop()
```

Medium

[M-1] Unbounded for loop to check for duplicates in `PuppyRaffle::enterRaffle` can lead to denial of service (DoS)attack

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates.However, the longer the `PuppyRaffle::players` array is, the more checkd a new player will have to make. This means the gas costs for players who enter right when the raffle starts will be dramatically lower tha those who enter later. Every additional address in the `players` array, is an addtional check the loop will ahve to make.

```
//@audit DoS Attack
@>    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");
        }
    }
```

Impact: The gas costs for raffle entrance will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of a raffle to be one of the first entrants in the queue.

An attacker might make the `PuppyRaffle::players` array so big, that no one else enters,guaranteeing themselves the win.

Proof of code:

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

- 1st 100 players: ~6252047 gas
- 2nd 100 players: ~18068137 gas

This is more than 3x more expensive for the second 100 players.

▶ PoC

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

        uint256 playerNum = 100;
        address[] memory players = new address[](playerNum);
        for (uint256 i = 0; i < playerNum; i++) {
            players[i] = address(uint160(i));
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playerNum}(players);
        uint256 gasEnd =gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players: ", gasUsedFirst);

        //second 100 players
        address[] memory playersTwo = new address[](playerNum);
        for (uint256 i = 0; i < playerNum; i++) {
            playersTwo[i] = address(uint160(i + playerNum));
        }
        uint256 gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playerNum}(playersTwo);
        uint256 gasEndTwo =gasleft();
        uint256 gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice;
        console.log("Gas cost of the second 100 players: ", gasUsedSecond);
        assert(gasUsedSecond > gasUsedFirst);
    }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

```
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;
    .
    .
    .
    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]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
        }

-        // Check for duplicates
+       // Check for duplicates only from the new players
+       for (uint256 i = 0; i < newPlayers.length; i++) {
```

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

3. Alternatively, you could use **OpenZeppelin's EnumerableSet library**.

[M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

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

Impact: This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 totalFees.
2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on the `PuppyRaffle::withdrawFees` function.

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

[M-3] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` there 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. There is a comment which says:

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

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

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: 
Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, 
block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

[M-4] Smart contract wallets raffle winners without a `receive` or a `fallback` function 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 wallets that rejects payment, the lottery would not be able to restart.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making a lottery reset difficult.

Also, true winners would not get paid out and someone else could take their money.

Proof of Concept:

1. 10 smart contract wallets enter the lotter 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 wallets entrants (not recommended)

2. Create a mapping of addresses -> payout amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended)

> Pull over push

Low

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

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

```
function getActivePlayerIndex(address player) external view returns (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }

@>      return 0;
    }
```

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

Proof of Concept:

1. User enters the raffle, they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the function documentation

Proof of Code:

▶ Code

Place the following into `PuppyRaffleTest.t.sol`

```
function testGetActivePlayerIndexEdgeCase() public {
        address[] memory players = new address[](2);
        players[0] = playerOne;
        players[1] = playerTwo;
        puppyRaffle.enterRaffle{value: entranceFee * 2}(players);

        assertEq(puppyRaffle.getActivePlayerIndex(playerOne), 0);
        assertEq(puppyRaffle.getActivePlayerIndex(address(3)), 0);
        console.log("playerOne Index:",
puppyRaffle.getActivePlayerIndex(playerOne));
        console.log("Inactive player Index:",
puppyRaffle.getActivePlayerIndex(address(3)));
    }
```

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

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active;

Gas

[G-1] Unchanged state variable should be declared as immutable or constant.

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

Instances:

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

[G-2] Storage variables in a loop should be cached.

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

```diff
+   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");
        }
    }
```

Recommendation: Cache the lengths of storage arrays if they are used and not modified in for loops

Informational / Non-Critical

[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;`

▶ 1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6; //q why less than v8? ownable is set by default
```

[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 a recent version of Solidity (at least 0.8.0) with no known severe issues.

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please see [slither] (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity) documentation for more information.

[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.

▶ 2 Found Instances

- Found in src/PuppyRaffle.sol Line: 66

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 204

  ```
          feeAddress = newFeeAddress;
  ```

[I-4] `PuppyRaffle::selectWinner` should follow CEI, which is a best practice

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

```diff
-       (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-5] Use of "magic" numbers is dicouraged

It can be confusing to see number literals in a codebase, and it's much more readable if the numbes are given a name.

Examples:

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

Instead, you could use:

```
    uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
    uint256 public constant FEE_PERCENTAGE = 20;
    uint256 public constant PRIZE_POOL_PRECISION = 100;
```

[I-6] State changes are missing events

[I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed