
Puppy Raffle Audit Report

palefish997



9 December 2025

Contents

Puppy Raffle Audit Report	3
Functionality	3
About palefish997	3
Disclaimer	3
Risk Classification	4
Audit Details	4
Scope	4
Protocol Summary	4
Roles	4
Executive Summary	5
Issues found	5
Findings	5
High	5
[H-01] Reentrancy in PuppyRaffle::refund allows entrant to drain contract balance	5
[H-02] Weak & Manipulable Randomness, The Raffle Outcome Can be Manipulated by External Parties	6
[H-03] The Contract Does a External Call in PuppyRaffle::selectWinner meaning it can fall victim of DoS Attack	6
[H-04] Integer overflow of PuppyRaffle::totalFees loses fees	7
Medium	8
[M-01] Looping Through Players Array in PuppyRaffle::enterRaffle is Potential DOS Vector if Large Number of Players are Added	8
[M-02] The PuppyRaffle::refund Creates Zero-address Holes in players Array, Allowing address (0) to Win the Raffle	9
[M-03] Balance check on PuppyRaffle::withdrawFees enables griefers to self-destruct a contract to send ETH to the raffle, blocking withdrawals	10
Low	11
[L-01] No Zero-Address / Zero-Value Checks in Constructor	11
[L-02] Outdated Solidity & OpenZeppelin Versions	11
[L-03] Rarity thresholds are off-by-one, causing 1% deviation in actual probabilities . . .	11

Informational	13
[I-01] Missing Naming Conventions	13
[I-02] Gas Optimizations	13
[L-03] Magic Numbers	13
[I-04] Dead Function PuppyRaffle::isActivePlayer	13
[I-05] Obsolete Function PuppyRaffle::getActivePlayerIndex	13
[I-06] Test Coverage	14
[I-07] Unchanged variables should be constant or immutable	14

Puppy Raffle Audit Report

Prepared by: palefish997

Auditor:

- palefish997

Functionality

Player: Player can ether raffle with a list of addesses, no duplicates allowed. Player can refund ant hey get their `value` back.

Owner: The contract deployer. Can set fee address through the `changeFeeAddress` function.. Gets but of `value`.

About palefish997

Aspiring independent smart contract security researcher and consultant. Actively developing deep expertise in Solidity, Ethereum security, and vulnerability detection through hands-on approach and continuous learning.

Disclaimer

The palefish997 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 palefish997 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

Audit Details

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

```
1 22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

```
1 ./src/
2 -- PuppyRaffle.sol
```

Protocol Summary

Puppy Raffle is a protocol dedicated to raffling off puppy NFTs with varying 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

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	3
Info	7
Total	17

Findings

High

[H-01] Reentrancy in PuppyRaffle::refund allows entrant to drain contract balance

Description: The `PuppyRaffle::refund` function does not follow CEI pattern, allowing reentrancy and drain the funds in the contract.

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

```
1 payable(msg.sender).sendValue(entranceFee);
2 players[playerIndex] = address(0); // state update AFTER send
```

Impact: An attacker contract can re-enter `PuppyRaffle::refund` function multiple times and drain more than their entrance fee.

Recommended Mitigation: Follow Checks-Effects-Interactions pattern. Update the `players` array before making the external call. Additionally, we should move the event emission up as well.

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

[H-02] Weak & Manipulable Randomness, The Raffle Outcome Can be Manipulated by External Parties

Description: Winner and rarity are derived from: `msg.sender` (the caller of `selectWinner`) and `block.timestamp/difficulty` are controllable/manipulable by miners or the caller.

Impact: Anyone can brute-force calls off-chain until they become the winner or get legendary rarity.

Recommended Mitigation: Use Chainlink VRF or at minimum commit-reveal scheme.

[H-03] The Contract Does a External Call in PuppyRaffle::selectWinner meaning it can fall victim of DoS Attack

Description: The contract does a external call in `PuppyRaffle::selectWinner`, if the winner is a malicious contract or simply does not have `receive` or `fallback` function, it becomes impossible to send the price.

```
1 solidity(bool success,) = winner.call{value: prizePool}("");  
2 require(success, "...");
```

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.

Recommended Mitigation: Use pull-over-push:

```
1 mapping(address => uint256) public pendingWithdrawals;  
2  
3 pendingWithdrawals[winner] += prizePool;  
4  
5 function withdrawPrize() external {  
6     uint256 amount = pendingWithdrawals[msg.sender];  
7     require(amount > 0);  
8     pendingWithdrawals[msg.sender] = 0;  
9     payable(msg.sender).transfer(amount);  
10 }
```

Pull-over-push pattern stores winnings in a mapping and allows user to withdraw themselves.

[H-04] Integer overflow of PuppyRaffle::totalFees loses fees

Description: In Solidity versions prior to 0.8.0, integers were subject to integer overflows.

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

Impact: In PuppyRaffle::selectWinner, totalFees are accumulated for the feeAddress to collect later in 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 first conclude a raffle of 4 players to collect some fees.
2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well.
3. totalFees will be:

```
1 totalFees = totalFees + uint64(fee);
2 // substituted
3 totalFees = 80000000000000000000 + 1780000000000000000000000;
4 // due to overflow, the following is now the case
5 totalFees = 153255926290448384;
```

4. You will now not be able to withdraw, due to this line in PuppyRaffle::withdrawFees:

```
1 require(address(this).balance ==
2     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 what the protocol is intended to do.

Recommended Mitigation:

Upgrade to Solidity >=0.8.0 for automatic overflow checks (reverts on overflow). Use uint256 for totalFees to avoid the cast.

Remove the balance check in PuppyRaffle::withdrawFees

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

Medium

[M-01] Looping Through Players Array in PuppyRaffle::enterRaffle is Potential DOS Vector if Large Number of Players are Added

Description: The PuppyRaffle::enterRaffle function loops through the `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 stats 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.

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::entrants array so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept: If we have 2 sets of 100 players, the gas consts will be as such:

- Gas used to enter many players: 23990872 gas
- Gas used to enter many players again: 88951944 gas

```
1  function test_denialOfService() public {
2      vm.txGasPrice(1);
3
4      uint256 playersNumber = 100;
5
6      // First time entering 100 players
7      address[] memory manyPlayers = new address[](playersNumber);
8      for (uint256 i = 0; i < playersNumber; i++) {
9          manyPlayers[i] = address(100 + i);
10     }
11     uint256 gasStart = gasleft();
12     puppyRaffle.enterRaffle{value: entranceFee * playersNumber}(
13         manyPlayers
14     );
15     uint256 gasEnd = gasleft();
16     console.log("Gas used to enter many players: ", (gasStart -
17         gasEnd));
18
19     // Second time entering 100 players
20     address[] memory manyPlayers2 = new address[](playersNumber);
21     uint256 gasStart2 = gasleft();
22     for (uint256 i = 0; i < playersNumber; i++) {
23         manyPlayers2[i] = address(200 + i);
24     }
```

```

24         puppyRaffle.enterRaffle{value: entranceFee * playersNumber}(
25             manyPlayers2
26         );
27         uint256 gasEnd2 = gasleft();
28         console.log(
29             "Gas used to enter many players again: ",
30             (gasStart2 - gasEnd2)
31         );
32         assert((gasStart - gasEnd) < (gasStart2 - gasEnd2));
33     }

```

Recommended Mitigation: Use Address to Index mapping, and check for duplicates in the first for loop.

```

1 +     mapping(address => uint256) public playerIndex;
2
3     function enterRaffle(address[] memory newPlayers) public payable {
4         require(
5             msg.value == entranceFee * newPlayers.length,
6             "PuppyRaffle: Must send enough to enter raffle"
7         );
8
9         for (uint256 i = 0; i < newPlayers.length; i++) {
10 +             require(players[playerIndex[newPlayers[i]]] != newPlayers[i]
11 +                 , "PuppyRaffle: Duplicate player")
12 +             players.push(newPlayers[i]);
13 +             playerIndex[newPlayers[i]] = players.length;
14         }
15 -         for (uint256 i = 0; i < players.length - 1; i++) {
16 -             for (uint256 j = i + 1; j < players.length; j++) {
17 -                 require(
18 -                     players[i] != players[j],
19 -                     "PuppyRaffle: Duplicate player"
20 -                 );
21 -             }
22 -         }
23         emit RaffleEnter(newPlayers);
24     }

```

[M-02] The PuppyRaffle::refund Creates Zero-address Holes in players Array, Allowing address(0) to Win the Raffle

Description: The current `PuppyRaffle::refund()` function removes a player by setting their position to `address(0)`. This leaves gaps (zero addresses) in the `players` array instead of actually removing the element. There is now a non-zero chance that `address(0)` is selected as the winner — especially after several refunds.

Impact: The `PuppyRaffle::selectWinner` function can pick a zero address as a winner.

Recommended Mitigation: Use swap-remove pattern.

The swap-remove pattern swaps the given index with the last place and pops the last place in the array and update the mapping. Refactored function with the `playerIndex` mapping, without the need of `PuppyRaffle::getActivePlayerIndex` function to call it.

```
1 mapping(address => uint256) public playerIndex;
2
3 function refund(address player) public {
4     require(player == msg.sender, "...");
5
6     uint256 index = playerIndex[player];
7
8     players[index] = players[players.length - 1];
9     playerIndex[players[index]] = index;
10    players.pop();
11
12    payable(msg.sender).sendValue(entranceFee);
13 }
```

[M-03] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdestruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

```
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 }
```

Impact: This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. PuppyRaffle has 800 wei in its balance, and 800 totalFees.

2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```

1   function withdrawFees() external {
2     -   require(address(this).balance == uint256(totalFees), "
3       PuppyRaffle: There are currently players active!");
4     uint256 feesToWithdraw = totalFees;
5     totalFees = 0;
6     (bool success,) = feeAddress.call{value: feesToWithdraw}("");
7     require(success, "PuppyRaffle: Failed to withdraw fees");
8 }
```

Low

[L-01] No Zero-Address / Zero-Value Checks in Constructor

Description: The `feeAddress` and `_entranceFee` cannot be set after contract is deployed, use zero address checks to prevent contract becoming accidentally unusable.

Recommended Mitigation: Use zero address checks.

```

1 require(_entranceFee > 0, "Entrance fee zero");
2 require(_feeAddress != address(0), "Fee address zero");
3 require(_raffleDuration > 0, "Duration zero");
```

[L-02] Outdated Solidity & OpenZeppelin Versions

The code style and some patterns suggest Solidity <0.8.0 (no built-in overflow checks) and old OpenZeppelin imports. Upgrade to the latest stable Solidity 0.8.18 version, and latest OpenZeppelin version.

[L-03] Rarity thresholds are off-by-one, causing 1% deviation in actual probabilities

Description: In the `PuppyRaffle::selectWinner` function generates a pseudo-random `rarity` value as follows:

```

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

This produces a uniformly distributed integer in the range [0, 99] (100 possible outcomes). However, the subsequent rarity checks use inclusive upper bounds (\leq), which creates an off-by-one error:

```

1 if (rarity  $\leq$  70)           // Common (intended: 70%)
2 else if (rarity  $\leq$  95)    // Rare   (intended: 25%)
3 else                      // Legendary (intended: 5%)

```

Because the maximum value is 99, the actual probability distribution becomes:

Rarity	Intended Chance	Actual Range	Actual Chance	Deviation
Common	70%	0–70	71%	+1%
Rare	25%	71–95	25%	0%
Legendary	5%	95–99	4%	-1%

This means common puppies are 1% more likely, and legendary puppies are 1% less likely than documented or intended.

Impact: Users are misled about the true odds of receiving a legendary puppy. The rarest tier (legendary) is slightly harder to obtain than advertised.

Recommended Mitigation: Fix the boundaries by using strict less-than comparisons ($<$). This correctly maps the 100 possible outcomes to the intended percentages:

```

1 -  if (rarity  $\leq$  COMMON_RARITY) {
2 +  if (rarity  $<$  COMMON_RARITY) {
3     tokenIdToRarity[tokenId] = COMMON_RARITY;
4 -  } else if (rarity  $\leq$  COMMON_RARITY + RARE_RARITY) {
5 +  } else if (rarity  $<$  COMMON_RARITY + RARE_RARITY) {
6     tokenIdToRarity[tokenId] = RARE_RARITY;
7 } else {
8     tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
9 }

```

Fixed probabilities:

Rarity	Range Values	Probability
Common	0–69	70%
Rare	70–94	25%
Legendary	95–99	5%

Alternatively, you could do `rarity = (hash % 100) + 1` and keep the original `<=` checks (shifting range to 1–100), but using `<` is the cleaner and more common approach.

Informational

[I-01] Missing Naming Conventions

Storage variables lack correct prefixes (`s_`, `i_`). This is a widely adopted best practice that reduces bugs.

[I-02] Gas Optimizations

Cache `players.length` in memory variables, to reduce gas fees.

[L-03] Magic Numbers

Replace 80, 20, 100, 4 (min players), etc. with clearly named constants.

[I-04] Dead Function `PuppyRaffle::_isActivePlayer`

`PuppyRaffle::_isActivePlayer` is never used, remove to reduce contract size.

[I-05] Obsolete Function `PuppyRaffle::getActivePlayerIndex`

`PuppyRaffle::getActivePlayerIndex` is no more used, remove to reduce contract size.

[I-06] 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)

[I-07] Unchanged variables should be constant or immutable

Constant Instances:

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

Immutable Instances:

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