# PuppyRaffle Protocol Audit Report

Prepared by:Ola Hamid

Date: 3rd September 2024

# Table of Contents

## Protocol Summary

This Protocol 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

I Ola Hamid 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 |

I 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

### Issues found

| Severity | Number of issues found |
|---|---|
| High | 4 |
| Medium | 3 |
| Low | 1 |
| Info | 9 |
| Gas | 4 |
| Total | 17 |

## Findings

### High Issues

#### [H-1] RE-Entrancy Attack in `PupyRaffle:Refund` allow entract to drain contact balance

**Description:** the `PupyRaffle:Refund` function does not follow a CEI(checks, effect and interactions), as a result enable attacker to drain the puppy raffle.

in the `PupyRaffle:Refund` function make an external call to the `msg.sender`, and only after that call do we update the array in the refund function.

```
    function refund(uint256 playerIndex) public {
        // @audit-skipped MEV ATTACK IN HERE --RECON
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the player
can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player already
refunded, or is not active");
        ///@audit re entrancy
@>        payable(msg.sender).sendValue(entranceFee);
@>       players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

An attacker can call the `fallback/receive` function that calls the `PupyRaffle:Refund` function, and do the same thing over and over again to drain the whole contract. **Impact:** All the fund paid by the players could be stolen by the malicious actor.

**Proof of Concept:**

1. users enter the raffle
2. attaacker set up a `fallback` and `receive` attack contact to call the `fallback/receive` function
3. attacker enters the raffle.
4. attacker calls the `fallback/receive` function, using the attack contract he created and the stealing the funds.

▶ CODE

```
  function testCheckReentrancy() public playerEntered {
         address[] memory players = new address[](3);
         players[0] = playerOne;
         players[1] = playerTwo;
         players[2] = playerOne;
         vm.expectRevert("PuppyRaffle: Duplicate player");
         puppyRaffle.enterRaffle{value: entranceFee * 3}(players);

         reentrancyAttack = new ReentrancyAttack(puppyRaffle);
         address AttackingUser = makeAddr("attackingUser");

         vm.deal(AttackingUser, 1 ether);
         uint256 startingAttackingContract = 
address(reentrancyAttack).balance;
         uint256 startingRaffleContract = address(puppyRaffle).balance;

         vm.prank(AttackingUser);
         reentrancyAttack.attack{value: entranceFee}();
         // assertEq(address(puppyRaffle).balance, 0);
         console.log ("starting Attcking contract Balance:", 
startingAttackingContract);
         console.log ("starting Raffle Contract 
Balance:",startingRaffleContract);

         console.log ("ending attacker contract balance:", 
address(reentrancyAttack).balance);
         console.log ("ending Raffle Contract Balance:", 
address(puppyRaffle).balance);
         // vm.stopPrank();
     }
```

**Recommended Mitigation:** To prevent this, you need to follow CEI, first of all we should update the `player` array before calling the external call. additional, you need to call move the event emmission 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: `abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`

Use `abi.encode()` instead which will pad items to 32 bytes, which will prevent hash collisions (e.g. `abi.encodePacked(0x123,0x456)` => `0x123456` => `abi.encodePacked(0x1,0x23456)`, but `abi.encode(0x123,0x456)` => `0x0...1230...456`). Unless there is a compelling reason, `abi.encode` should be preferred. If there is only one argument to `abi.encodePacked()` it can often be cast to `bytes()` or `bytes32()` instead. If all arguments are strings and or bytes, `bytes.concat()` should be used instead.

▶ 2 Found Instances

- Found in src/PuppyRaffle.sol Line: 248

  ```
              abi.encodePacked(
  ```

- Found in src/PuppyRaffle.sol Line: 252

  ```
                      abi.encodePacked(
  ```

### [H-3] Weak Randomness in `PuppyRaffle::selectWinner` allow winner to influence or predict the winner and predict the winning puppy.

**Description:** hashing the `msg.sender` , `block.timestamp` and `block.difficulty` values creates a predictable numbers, malicious user can manipulate this values and choose the winner ahead of time

also, user can front run this act and check if they are the winner or not, if they aren't the winner they can call the refund ahead pf time **Impact:** Any user can influence the raffle, winning the money and selecting the rarest puppy making the entire raffle worthless ans know who wins ahead of time

**Proof of Concept:**

1. validators can know ahead of time the block.timestamp and block.difficulty and use that to predict to when/how to participate.
2. user caan mine/manipulate their `msg.sender` value to result their address being used to generate and manipulate the winner!
3. users can revert their `revert` thier selectWinner function if they ont like the winner or resulting puppy. **Recommended Mitigation:** consider using a cryptographically provable random number generator such as chainlink VRF

### [H-4] Interger overflow of `puppyRaffle::totalFee` loses fees

**Description:** In solidity versions prior to `0.8.0` intergers were subject to intergers overflows

```
uint64 myVar = type(uint64).max;
// 18446744073709551615
uint64 myVar = myVar + 1
// 0
```

**Impact:** in the function `PuppyRaffle::selectWinner` is collecting the `puppyRaffle::totalFee`to be accumulated to the `FeeAddress` and collected later on in the `puppyRaffle::wiithdrawFee` function, if the `totalFee` overflows, the `FeeAddress` may not be collect the correct amount of Fee, leaving fees permanently stuck in the contract.

**Proof of Concept:**

1. alot of user call the enterRaffle function
2. pay alot of fee, alot more than the `puppyRaffle::totalFee` uint64 can take.
3. `totalFee` passes uint64.max
4. fee get stucked causing lack of funds.

▶ 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:**

1. use a newer version of solidity `0.8.24`
2. you could use safe Math library in OZ in `0.7.24` version of solidity, however you would still have a hard time with the `uint64` types if there are too many fees

## MEDIUM

### [M-1] Looping through the player array to check for duplicate in the `PuppyRaffle::enterRaffle` function, is a potential denial of service (DOS) attack, increamenting Gas cost for future entrants

**Description:** The `PuppyRaffle::enterRaffle` function Loop through the player array to check,for duplicates, however, the longer the `PuppyRaffle:: players` array is, the more the checks a new player will have to make. this means the gas cost for players who enter right when the rafflewill be drammatically lower than those who enter later. Every additonal address in the `players` array, is an additional check the loop here to make

**Impact:** The Gas cost on the `PuppyRaffle::enterRaffle` function will signifanctly grow high as more users enter the array `PuppyRaffle:: players` causing a high cost of gas to be highly spent and discoraging late users from comign into the raffle.

An attacker might be able to know how to get hold of the array, call it multiple times, fill it up not allowing an external user to come in and potentially giving himself the win

**Proof of Concept:** if you have 2 set of 100 players, the gas cost will be as such:

first 100 players: `20376518` second 100 players: `67687279`

Add the following test code to the your `PuppyRaffle.t.sol` test contract.

▶ CODE

```
    function testDOSInEnterRaffle() public {
        //ARRANGE
        //LETS ENTER 100 PLayers
        // vm.startPrank(User1);
        // what the act is doing is that, it is assuming that 100 users
are coming to enter the raffle
        vm.txGasPrice(1);

        uint playersNum = 200;
        address[] memory players = new address[](playersNum);
        for (uint i = 0 ; i < playersNum; ++i) {
            players[i] = address(i);
        }
        // ACT
        //now,we are letting the 100 players enter the raffle FOR ACTING.
        uint gasStart = gasleft();
        puppyRaffle.enterRaffle{value:entranceFee * players.length}
(players);
        uint gasEnd = gasleft();
        uint gasUsed = (gasStart - gasEnd) * tx.gasprice;

        console.log("gas cost of the current gas used:", gasUsed);


        address[] memory players2 = new address[](playersNum);
        for (uint i = 0 ; i < playersNum; i++) {
            players2[i] = address(i + playersNum);
        }
        // ACT
        //now,we are letting the 100 players enter the raffle FOR ACTING.
        uint gasStart2 = gasleft();
        puppyRaffle.enterRaffle{value:entranceFee * players2.length}
(players2);
        uint gasEnd2 = gasleft();
        uint gasUsed2 = (gasStart2 - gasEnd2) * tx.gasprice;

        console.log("gas cost of the second current gas used:", gasUsed2);
    }
```

**Recommended Mitigation:** There are a feew recommendations.

1. consider Allowing Duplicate: User can make a new addresses anyways, so the check doesnt prevent the user from entering multiple times only prevent same addresses. so the check doesnt really check for duplicates.

2. Consider using a mapping to check for duplicate, this will allow constant time lookUp of whether a user has already entered.

```
+     mapping (address => bool) public raffleAddrExists;
```

```
    .
    .
    .
        function enterRaffle(address[] memory newPlayers) public payable {
        //q were custom revert a thing then while it was 0.7.4 solidity
version?
        //q what if it is zero?

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

-        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");
-            }
-        }
+        for (uint256 i = 0; i < players.length; ++i) {
+            if (raffleAddrExists[players[i]] == true ) {
+                revert raffle_duplicateError();
+            }
+        }
+        emit RaffleEnter(newPlayers);
+    }
```

### [M-2] Unsafe Cast of `PuppyRaffle::Fee` loses fees

**Description:** In `PuppyRaffle::selectWinner` there is an unsafe typecast of `PuppyRaffle::Fee` from uint256 to uint64, if the value of the uint256 is larger than that of the `type(uint64).max`, the value will be truncated.

```
function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
players");
        uint256 winnerIndex =
uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
        totalFees = totalFees + uint64(fee);
@>        uint256 tokenId = totalSupply();
.......
}
```

**Impact:** the `PuppyRaffle::selectWinner` contains a casting for the `PuppyRaffle::totalFee` with the `PuppyRaffle::Fee` converting the uint256 to uint64. the amount of `PuppyRaffle::Fee` can be alot higher than the max amount uint64 can take since it is a unit256 causing loss of funds.

**Proof of Concept:**

1. 100 users call the enter the raffle, passing the max amount of the uint64.
2. The `PuppyRaffle::Fee` is stored as `50000000000000000000` since it has a uint 256.
3. casting `50000000000000000000` to uint64 will result to smaller causing loss of funds

▶ 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:**

1. use a newer version of solidity `0.8.24`
2. you could use safe Math library in OZ in `0.7.24` version of solidity, however you would still have a hard time with the `uint64` types if there are too many fees
3. cast totalFee to `uint256` instead of `uint64`.

### [M-3] Smart contract wallets raffle without a receive and fallback function will block the start of a new contest

**Description:** The `PuppuyWinner::selectWinner` function is responsible for resetting the lottery, however, if the winner is a smart contract wallet that rejects payments, the lottery will not be reset to get restarted

users could call the `selectWinner` again and the non-wallet entrants could cost a lot due to the deuplicate check and a lottery reset could get very challenging **Impact:** The `PuppuyWinner::selectWinner` function could revert many times, making the lottery reset very challenging

Also, true winner wil not get paid out, and someone else could take thier money **Proof of Concept:**

1. 10 smart contract wallets enter the lottery without a fallback or receive function.
2. The lottery ends
3. The selectWinner function wouldn't work, even though the lottery is over!

**Recommended Mitigation:**

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, putting the owness on the winner to claim their prize. (Recommended) i.e PULL OVER PUSH

## LOW

### [L-1] `PupyRaffle:getActivePlayerIndex` function returns 0 for inExistence players in the loop and for players at index, causing a player at index 0 to think he has not entered the raffle

**Description:** if the player is at index zero it will return zero and the player will think they are not in the raffle

```
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 will think he has not entered the raffle, attemting to enter the raffle again wasting too muc gass

**Proof of Concept:**

1. users enter the raffle, they are the first entrant
2. `PupyRaffle:getActivePlayerIndex` returns 0
3. user think they havnet entered correcntly due to the function documentation

**Recommended Mitigation:** the easiest way is to revert the if statement if the playing isnt in the raffle

## INFORMATIONALS

### I-1: Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

▶ 2 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

- Found in src/puppyAttack.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

### I-2: Using an outdated version of solidity isn't recommended

Solc frequently releases new compiler version. using san old version prevents access to new solidity security checks. **Reccomendation**

Upgrading to Solidity `0.8.24` is recommended to enhance security, introduce new features, resolve bugs, and improve performance. The newer version includes built-in overflow and underflow checks, new language features like custom errors and try/catch, and various optimizations for more efficient contract execution.

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.

▶ 3 Found Instances

- Found in src/PuppyRaffle.sol Line: 67

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 214

```
        feeAddress = newFeeAddress;
```

- Found in src/puppyAttack.sol Line: 11

```
        puppyRaffle = _puppyRaffle;
```

### I-6: Event is missing `indexed` fields

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.

▶ 3 Found Instances

- Found in src/PuppyRaffle.sol Line: 54

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

- Found in src/PuppyRaffle.sol Line: 55

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

- Found in src/PuppyRaffle.sol Line: 56

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

### I-7: Define and use `constant` variables instead of using literals

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

▶ 3 Found Instances

- Found in src/PuppyRaffle.sol Line: 162

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

- Found in src/PuppyRaffle.sol Line: 163

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

- Found in src/PuppyRaffle.sol Line: 175

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

### [I-8] `PuppyRaffle:selectWinner` does not follow CEI, which is not a best practice

It's best to keep code clean and follow CEI, checks, effect and interactions

```diff
-        (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-9] MAGIC NUMBER Use of magic numbers have been highly discouraged as it cam hard t0 read

EXAMPLES;

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

Instead, you could use:

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

# GAS

### [G-1] UNCHANGED STATE VARIABLE TO BE DECLARED CONSTANT OR IMMUTABLE

Reading from storage is much more expensive than reading from a contant or immutable.

Instances:

`PuppyRaffle:raffleDuration` should be `immutable`

`PuppyRaffle:commonImageUri` should be `constant`

`PuppyRaffle:rareImageUri` should be `constant`

`PuppyRaffle:legendaryImageUri` should be `constant`

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

Everytime you call player.length you read from storage as to memory which is alot more cheaper and more efficient

```
+    uint 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 < players.length; j++) {
+            for (uint256 j = i + 1; j < playersLength; j++) {
                require(players[i] != players[j], "PuppyRaffle: Duplicate
player");
            }
```

## G-3: `public` functions not used internally could be marked `external`

Instead of marking a function as `public`, consider marking it as `external` if it is not used internally.

▶ 3 Found Instances

- Found in src/PuppyRaffle.sol Line: 87

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

- Found in src/PuppyRaffle.sol Line: 116

  ```
      function refund(uint256 playerIndex) public {
  ```

- Found in src/PuppyRaffle.sol Line: 240

```
    function tokenURI(uint256 tokenId) public view virtual override
returns (string memory) {
```

### G-4: Loop contains `require`/`revert` statements

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

▶ 1 Found Instances

- Found in src/PuppyRaffle.sol Line: 99

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

### [G-5] _isActivePlayer is never used and should be removed

The function PuppyRaffle::_isActivePlayer is never used and should be removed.

```diff
-    function _isActivePlayer() internal view returns (bool) {
-        for (uint256 i = 0; i < players.length; i++) {
-            if (players[i] == msg.sender) {
-                return true;
-            }
-        }
-        return false;
-    }
```