Protocol Audit Report

Version 1.0

Cyfrin.io

May 3, 2025

Protocol Audit Report

Cyfrin.io

March 7, 2023

Prepared by: Cyfrin Lead Auditors: - xxxxxxx

Table of Contents

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 Cyfrin 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

The Audit if done for 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 This was my fist audit and I use static Analysis tool Slither in this. ## Issues found

Findings

High

[H-3] Integer Overflow in `PuppyRaffle::totalFees` which will cause lose funds

Description: The `PuppyRaffle::totalFees` is uint64 storage variable that can store a max value 18.44 ether. If the value exceeds this max value then value will roll back to 0 again. Suppose we store 20 eth in this, then the remaining value 20eth-18.44 ether will be stored in the variable.

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

Impact: The Raffle totalFees funds will overflow and funds will permanently get stucked.

Proof of Concept: We enter the raffle with 100 players. And then we select a winner using the `PuppyRaffle::selectWinner` function. According to natspec, the totalFees will be 20% of total

funds(100eth) which will be 20 ether. But totalFees is way less than this(around 1.55 eth only) due to uin64 overflow.

Due to funds loss, the withdraw function won't work because of this line:

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

Although we can use `selfdestruct` to send ether to this contract to match the values for withdraw, but this is not the intended use of the protocol.

POC

```
function testTotalFeesOverflow() public{
    uint256 playersLen=100;
    address[] memory players=new address[](playersLen);
    for(uint256 i=0; i<playersLen; i++){
        players[i]=address(i);
    }
    address user=makeAddr("user");
    vm.deal(user,100 ether);
    vm.prank(user);
    puppyRaffle.enterRaffle{value:playersLen*entranceFee}(players);

    vm.roll(block.number+1);
    vm.warp(block.timestamp+duration);
    puppyRaffle.selectWinner();
    //totalFees should be 20 eth, but it is only 1.55 ether.
    assert(puppyRaffle.totalFees() < 1.6 ether);
}
```

Recommended Mitigation: There are a few recommendations: 1. Use a bigger size variable like `uint256` to store totalFees funds and latest solidity version (atleast `0.8.0`) so that overflow checks can happen. 2. You can use SafeMath Library of OpenZeppelin for solidity version 0.7.6 but still will have issue storing in uint64 if too much fess is collected. 3. Remove the match condition check in withdraw function. `diff - require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");` There are more attack vectors with that final require check so we recommend removing it.

Medium

[M-1] `PuppyRaffle::enterRaffle` Function is looping overa unbounded Players array that causes Denial Of Service to later users

Description: The `PuppyRaffle::enterRaffle` function loops over the `PuppyRaffle::players` to check for duplicates. This means a player entered at start will have very relatively low gas

cost rather than a person entering raffle at end. And this gas will keep increasing as more users enter raffle making ultimately a Denial Of Service(DoS).

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

Impact: As more and more users enter the raffle, the gas will keep increasing making the enterin raffle literally impossible to huge gas. An Attacker can make this array so big that no other users can participate in this raffle.

Proof of Concept:

If a user enter the raffle at start and a user enter the raffle after 100 people entered the raffle: - 1st player : ~61585 gas - 101th player(after 100 people entered): ~4361831 gas

The gas increased by 70 times for 101th user compared to 1st user

PoC

Place the following test in `PuppyRaffleTest.t.sol`:

```
function testEnterFaffleCauseDos() public{
        //1st person entering the raffle
        address[] memory user1=new address[](1);
        user1[0]=address(100);
        uint256 gasStart1=gasleft();
        puppyRaffle.enterRaffle{value:entranceFee}(user1);
        uint256 gasCost1=gasStart1-gasleft();

        //100 people entering the raffle
        address[] memory user2=new address[](100);
        for(uint256 i=0; i<user2.length; i++){
            user2[i]=address(i);
        }
        puppyRaffle.enterRaffle{value:entranceFee*user2.length}(user2);

        //now 102th person entering the raffle
        address[] memory user3=new address[](1);
        user3[0]=address(101);
        uint256 gasStart2=gasleft();
        puppyRaffle.enterRaffle{value:entranceFee}(user3);
        uint256 gasCost2= gasStart2-gasleft();

        console.log(gasCost1,gasCost2);
        assert(gasCost2>gasCost1);
    }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowin Duplicates: Users can anyway create new wallets and a same person can enter raffle from multple wallets. Duplications prevents a same wallet entering the raffle and not a person.
2. Use Mapping for checking duplicates: This will make a constant lookup gas for checking duplicates.

```
1  for (uint256 i = 0; i < players.length - 1; i++) {
2  for (uint256 j = i + 1; j < players.length; j++) {
3      require(players[i] != players[j], "PuppyRaffle: Duplicate
       player");
4  }
5  }
```

```
1
2  +    mapping(address=>uint256) playersToRaffleId;
3  +    uint256 raffleId=0;
4  .
5  .
6  .
7      function enterRaffle(address[] memory newPlayers) public
       payable {
8          require(msg.value == entranceFee * newPlayers.length, "
           PuppyRaffle: Must send enough to enter raffle");
9          for (uint256 i = 0; i < newPlayers.length; i++) {
10             players.push(newPlayers[i]);
11             playersToRaffleId[newPlayers[i]]=raffleId;
12         }
13         // Check for duplicates
14 -        for (uint256 i = 0; i < players.length - 1; i++) {
15 -            for (uint256 j = i + 1; j < players.length; j++) {
16 -                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
17 -            }
18 -        }
19 +        for(uint256 i=0; i<newPlayers.length; i++){
20 +            if(players[newPlayers[i]]!=raffleId){
21 +                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
22 +            }
23 +        }
24         emit RaffleEnter(newPlayers);
25     }
26 .
27 .
28 .
```

```
29      function selectWinner() external {
30  +       raffleId=raffleId+1;
31          require(block.timestamp >= raffleStartTime +
                raffleDuration, "PuppyRaffle: Raffle not over");
```

3. Alternatively, you could use OpenZeppelin EnumerableSet library

[M-2] Unsafe casting of uint256

[M-3] Winner chosen from `PuppyRaffle::selectWinner` can be a smart contract which misses fallback and recieve function will not get prize money

Description: The `PuppyRaffle::selectWinner` function can select a winner who is a smart contract and that don't have a recieve/fallback method, then the sending prize transaction will get reverted. This can happen multiple time causing a high gas cost and real users might not get their dues.

Impact: It can take a very long time toselect a winner who can recieve prize funds and also waste a lot of gas.

Proof of Concept: - 5 smart contracts which lacks fallback/recieve methods enter the raffle. - The raffle ends. - The `PuppyRaffle::selectWinner` function won't be able to send prize funds to any winner as all players are smart contracts even though raffle has ended.

Recommended Mitigation: There are a few recommendations: 1. Do not allow smart contracts to enter the raffle(not recommended as multi sigs should be able to enter raffle) 2. PUSH OVER PULL : We can create a mapping(address=>payouts) and a `PuppyRaffle::claimPrize` function which allows winners to claim and withdraw funds rather than directly sending them.(recommended)

[M-4] `PuppyRaffle::withdrawFees` function has a condition check that Mishandles ETH which makes withdraw fees nearly impossible

Description: The `PuppyRaffle::withdrawFees` has a check that contract balance must be equal to `PuppyRaffle::totalFees` , else one can't withdraw the fees fund. This can be misused by bad playerss that they can enter raffle a raffle instantly so that the contract balance is always more than `PuppyRaffle::totalFees` and withdrawing fees will be not possible.

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

Impact: The Raffle Fees money can get stuck in contract always as `PuppyRaffle::withdrawFees` function is never executed.

Proof of Concept: - First 5 players enter the raffle. - Raffle duration is over and a winner is selected and the raffle is reset. - But before withdrawing the fees , if someone enters the raffle instantly then we can't withdraw the funds as contract balance is more than `PuppyRaffle::totalFees`.

POC

```
function testWithdrawFeesIsMishandlingEth() public
    fivePlayerEnterRaffle{
     vm.warp(block.timestamp+duration);
     vm.roll(block.number+1);
     puppyRaffle.selectWinner();
     assertEq(address(puppyRaffle).balance,puppyRaffle.totalFees());

     address[] memory players=new address[](1);
     players[0]=address(0);
     address sender=makeAddr("sender");
     vm.deal(sender, 5 ether);
     vm.prank(sender);
     puppyRaffle.enterRaffle{value:1 ether}(players);
     //now contract balance is 1 ether more than totalFees as 1 player
        entered raffle before withdrawing Fees
     assert(address(puppyRaffle).balance!=puppyRaffle.totalFees());
}
```

Recommended Mitigation: The condiiton check responsible for this issue must be removed. There is no need of that check in function.

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

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` function will always show inactive for 1st raffle player

Description: `PuppyRaffle::getActivePlayerIndex` function returns the index of player in the players array. If the player is not active or don't exist it will return 0 index. But that will create a issue for 0th index player. The function will return always 0 for 0th index even if he is active or inactive.

```
    function getActivePlayerIndex(address player) external view returns
        (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
    //0th index player will also return 0 making him inactive
```

```
        return 0;
    }
```

Impact: 0th index player will think himself as inactive even if he is active also.

Proof of Concept: If 2 players(user1 and user2) enters the raffle. Now players array has 2 length player. `PuppyRaffle::getActivePlayerIndex` function will return 0 for user1 showing him as inactive.

POC

```
function testZeroIndexPlayerIsShownInactiveAlways() public{
     address user1=makeAddr("user1");
     address user2=makeAddr("user2");
     vm.deal(user1,5 ether);
     address[] memory players=new address[](2);
     players[0]=user1;
     players[1]=user2;
     vm.prank(user1);
     puppyRaffle.enterRaffle{value:2 ether}(players);

     uint256 player1Ind=puppyRaffle.getActivePlayerIndex(user1);
     uint256 player2Ind=puppyRaffle.getActivePlayerIndex(user2);
     assertEq(player1Ind,0); //0 means player1 is not present in raffle
     assertEq(player2Ind,1); //1 means player2 is at index1 in raffle
}
```

Recommended Mitigation: There are a few recommendations: 1. Instead of returning 0 for inactive players, we should return some value that will never be returned(like negative numbers will never be returned for array index).We can use -1 here.

```
```diff
-    function getActivePlayerIndex(address player) external view
    returns (uint256) {
+    function getActivePlayerIndex(address player) external view
    returns (int256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
-        return 0;
+        return -1;
    }
```
```

2. We can revert with a custom error if the player is inactive or don't exists. "'diff

    - error PuppyRaffle__PlayerInactive(); function getActivePlayerIndex(address player) exter-

nal view returns (uint256) { for (uint256 i = 0; i < players.length; i++) { if (players[i] == player) { return i; } }

```
    revert PuppyRaffle__PlayerInactive();
```

} “‘

Informational

[I-1] Usage of Floating Solidity pragma version makes code prone to errors

Description: We should stick to a specific solidity version rather than a floating version to minimise errors that an cause by future solidity versions. Using a speciific version ensures predicitibility and security but a furure version after `0.7.6` might have untested bugs that can cause bugs in our code.

```
pragma solidity ^0.7.6;
```

Impact: Future version might have some bugs.

Recommended Mitigation: Use a specific solidity version like `pragma solidity 0.8.18`.

[I-2] Outdated Solidity version Usage can cause issues

Description: We are using a solidity version `0.7.6` which is very outdated and don't have latest features like Arithmetic overflow and underflow checks etc. We should latest tested solidity versions to avoid language specific errors.

Impact: Old bugs can come from outdated solidity versions.

Proof of Concept:

Recommended Mitigation: Use a latest version like `0.8.18`.

Please see slither documentation for more informtaion.

[I-3] Zero Address Checks for `PuppyRaffle::feeAddress` to avoid Null Address

Description: We should do input sanitisation for address to avoid null or empty address that users might accidently put. We should have zero address checks for `PuppyRaffle::feeAddress` in the constructor and `PuppyRaffle::changeFeeAddress` function.

```
constructor(uint256 _entranceFee, address _feeAddress, uint256
    _raffleDuration) ERC721("Puppy Raffle", "PR") {
    entranceFee = _entranceFee;
```

```
3       // @audit-info check for zero address , input validation
4       feeAddress = _feeAddress;
5   }
6
7   function changeFeeAddress(address newFeeAddress) external onlyOwner {
8       // @audit-info check for zero address , input validation
9       feeAddress = newFeeAddress;
10      emit FeeAddressChanged(newFeeAddress);
11  }
```

Impact: Zero address can be set in feeAddress and if someone calls withdraw then all fee Money will be sent to zero address(Loss of funds). Although owner can change the feeAddress anytime later also.

Proof of Concept:

Recommended Mitigation: We will have a zero address check before updating our feeAddress and revert a zeroAddress error if someone puts accidently a null address.

```
1   +   error PuppyRaffle__ZeroAddress();
2       constructor(uint256 _entranceFee, address _feeAddress, uint256
            _raffleDuration) ERC721("Puppy Raffle", "PR") {
3           entranceFee = _entranceFee;
4   +       if(_feeAddress==Address(0)){
5   +           revert PuppyRaffle__ZeroAddress();
6   +       }
7           feeAddress = _feeAddress;
8   }
9
10      function changeFeeAddress(address newFeeAddress) external onlyOwner
            {
11  +       if(_feeAddress==Address(0)){
12  +           revert PuppyRaffle__ZeroAddress();
13  +       }
14          feeAddress = newFeeAddress;
15          emit FeeAddressChanged(newFeeAddress);
16  }
```

[I-4] `PuppyRaffle::selectWinner` function doesn't follow CEI Pattern

Description: In any function CEI Pattern must be followed means there should be checks at the first then there should be Effects(contract state changes) and at last should be Interactions(external contract calls).

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

```
5 +     require(success, "PuppyRaffle: Failed to send prize pool to winner
  ");
```

[I-5] Usage of Maic Numbers should be discouraged.

Description: We should not use literal values and the code is more readble if there are names used to literal values.

```
1 -     uint256 prizePool = (totalAmountCollected * 80) / 100;
2 -     uint256 fee = (totalAmountCollected * 20) / 100;
3 +     uint256 public constant TOTAL_POOL_SHARE=100;
4 +     uint256 public constant PRIZE_POOL_SHARE=80;
5 +     uint256 public constant FEE_POOL_SHARE=20;
6 +     uint256 prizePool = (totalAmountCollected * TOTAL_POOL_SHARE) /
  TOTAL_POOL_SHARE;
7 +     uint256 fee = (totalAmountCollected * FEE_POOL_SHARE) /
  TOTAL_POOL_SHARE;
```

[I-6] `PuppyRaffle::_isActivePlayer` function is never used anywhere and should be removed.

[I-7] `PuppyRaffle::FeeAddressChanged` event should have indexed address parameter for efficient logging.

Description:

```
1 -     event FeeAddressChanged(address newFeeAddress)
2 +     event FeeAddressChanged(address indexed newFeeAddress)
```

Gas

[G-1] Unchanged Variables should be marked immutable or constant to save Gas

Description: The Gas cost of constants and immutable variable are way less than storage variable as constants and immutables are directly stored in contract bytecode. Instances: - `PuppyRaffle::raffleDuration` should be `immutable`; - `PuppyRaffle::commonImageUri` should be `constant`; - `PuppyRaffle::rareImageUri` should be `constant`; - `PuppyRaffle::legendaryImageUri` should be `constant`;

Impact: It will save some gas, no major impact other than that.

Recommended Mitigation:

```
uint256 public immutable raffleDuration;
uint256 public constant commonImageUri;
uint256 public constant rareImageUri;
uint256 public constant legendaryImageUri;
```

[G-2] Array length in a loop should be cached to save gas.

Description: It is more gas efficient to cache the array length in a loop in a local memory variable rather than directly using the storage variable .

Recommended Mitigation:

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

+       uint256 playerLen=players.length;
+       for (uint256 i = 0; i < playerLen - 1; i++) {
-       for (uint256 i = 0; i < players.length - 1; i++) {
+           for (uint256 j = i + 1; j < playerLen; j++) {
-           for (uint256 j = i + 1; j < players.length; j++) {
                require(players[i] != players[j], "PuppyRaffle:
                   Duplicate player");
            }
        }
        emit RaffleEnter(newPlayers);
    }
.
.
.
    function getActivePlayerIndex(address player) external view returns
        (uint256) {
+       uint256 playersLen=players.length;
+       for (uint256 i = 0; i < playersLen; i++) {
-       for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
        return 0;
    }
```